

A GPU tour of wave propagation

Pengliang Yang

December 8, 2013

Contents

1 Basic acoustic wave equation	1
1.1 Taylor and Pade expansion	2
1.2 Approximate the wave equation	3
2 Forward modelling	3
2.1 Elastic wave equation	3
2.2 Absorbing boundary condition (ABC)	3
2.3 Perfectly Matched Layer (PML)	4
2.3.1 Split PML (SPML) for acoustics	4
2.3.2 Nonsplit Convolutional-PML (CPML) for acoustics	4
2.3.3 Non-split PML (NPML) for elastics	5
2.4 Discretization	6
2.4.1 Higher-order approximation of staggered-grid finite difference	6
2.4.2 Discretization of SPML	7
2.4.3 Discretization of NPML	8
3 Reverse time migration (RTM)	10
3.1 Features and benefits of RTM	10
3.2 RTM implementation	10
3.3 Imaging condition	10
4 Full waveform inversion (FWI)	12
4.1 The Newton, Gauss-Newton, and steepest-descent methods	13
4.2 Conjugate gradient (CG) implementation	14
4.3 CG method for iteratively reweighted least squares (IRLS)	15
4.4 Fréchet derivative	16
4.5 Gradient computation	17
A Forward modelling with PA: CUDA code	17
B Compute 2N-order difference coefficients: MATLAB code	19
C RTM: CUDA code	19

1 Basic acoustic wave equation

Define $\mathbf{x} = (x, y, z)$, time t , the s -th energy source function $\tilde{S}(\mathbf{x}, t; \mathbf{x}_s)$, pressure $p(\mathbf{x}, t; \mathbf{x}_s)$, particle velocity $\mathbf{v}(\mathbf{x}, t)$, material density $\rho(\mathbf{x})$, the bulk modulus $\kappa(\mathbf{x})$. Now we have

- Newton's law

$$\rho(\mathbf{x}) \frac{\partial \mathbf{v}(\mathbf{x}, t; \mathbf{x}_s)}{\partial t} = \nabla p(\mathbf{x}, t; \mathbf{x}_s). \quad (1)$$

- Constitutive law

$$\frac{1}{\kappa(\mathbf{x})} \frac{\partial p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t} = \nabla \cdot \mathbf{v}(\mathbf{x}, t; \mathbf{x}_s) + \tilde{S}(\mathbf{x}, t; \mathbf{x}_s). \quad (2)$$

Acoustics is a special case of gas dynamics (sound waves in gases and fluids) and linear elastodynamics. Note that elastodynamics is a more accurate representation of earth dynamics, but more industrial seismic processing based on acoustic model. Recent interest in quasiaoustic anisotropic approximations to elastic p-waves.

Assume $\tilde{S}(\mathbf{x}, t; \mathbf{x}_s)$ is differentiable constitutive law w.r.t. t . Substituting Eq. (2) into the differentiation of Eq. (1) gives

$$\frac{1}{\kappa(\mathbf{x})} \frac{\partial^2 p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} = \nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}, t; \mathbf{x}_s) \right) + \frac{\partial \tilde{S}(\mathbf{x}, t; \mathbf{x}_s)}{\partial t}. \quad (3)$$

We introduce $v(\mathbf{x}) = \sqrt{\frac{\kappa(\mathbf{x})}{\rho(\mathbf{x})}}$ (compressional p-wave velocity):

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} = \rho(\mathbf{x}) \nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}, t; \mathbf{x}_s) \right) + \rho(\mathbf{x}) \frac{\partial \tilde{S}(\mathbf{x}, t; \mathbf{x}_s)}{\partial t} \quad (4)$$

Under constant density condition, we obtain the 2nd-order equation

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} = \nabla^2 p(\mathbf{x}, t; \mathbf{x}_s) + f_s(\mathbf{x}, t; \mathbf{x}_s) \quad (5)$$

where $\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial z^2}$, $f_s(\mathbf{x}, t; \mathbf{x}_s) = \rho(\mathbf{x}) \frac{\partial \tilde{S}(\mathbf{x}, t; \mathbf{x}_s)}{\partial t}$. It is the simplest to understand and the basis for most processing (Here the spatial operator is spatially homogeneous). It is increasing popular because many anisotropic generalizations have been developed.

1.1 Taylor and Pade expansion

The Taylor expansion of a function $f(x+h)$ at x is written as

$$f(x+h) = f(x) + \frac{\partial f(x)}{\partial x} h + \frac{1}{2!} \frac{\partial^2 f(x)}{\partial x^2} h^2 + \frac{1}{3!} \frac{\partial^3 f(x)}{\partial x^3} h^3 + \dots \quad (6)$$

A popular example is

$$(1+x)^\alpha = 1 + \alpha x + \frac{\alpha(\alpha-1)}{2!} x^2 + \dots + \frac{\alpha(\alpha-1) \cdots (\alpha-n+1)}{n!} x^n + \dots \quad (7)$$

Here we mainly consider the following expansion formula:

$$(1-x)^{\frac{1}{2}} = 1 - \frac{1}{2}x - \frac{1}{8}x^2 - \frac{1}{16}x^3 - \frac{5}{128}x^4 - \dots, |x| < 1. \quad (8)$$

The Pade expansion of Eq. (8) should be:

$$(1-x)^{\frac{1}{2}} = 1 - \frac{x/2}{1 - \frac{x/4}{1 - \frac{x/4}{\dots}}} \quad (9)$$

we provide the derivation:

$$\begin{aligned} y = (1-x)^{\frac{1}{2}} &\Rightarrow x = 1 - y^2 = (1-y)(1+y) \Rightarrow 1-y = \frac{x}{1+y} \\ y = 1 - \frac{x}{1+y} &= 1 - \frac{x}{1 + (1 - \frac{x}{1+y})} = 1 - \frac{x/2}{1 - \frac{x/2}{1+y}} \\ &= 1 - \frac{x/2}{1 - \frac{x/2}{2 - \frac{x}{1+y}}} = 1 - \frac{x/2}{1 - \frac{x/4}{1 - \frac{x/2}{1+y}}} = \dots \end{aligned}$$

The 1st-order Pade expansion is:

$$(1-x)^{\frac{1}{2}} = 1 - \frac{x}{2} \quad (10)$$

The 2nd-order Pade expansion is:

$$(1-x)^{\frac{1}{2}} = 1 - \frac{x/2}{1 - \frac{x}{4}}. \quad (11)$$

And the 3rd-order one is:

$$(1-x)^{\frac{1}{2}} = 1 - \frac{x/2}{1 - \frac{x/4}{1 - \frac{x}{4}}}. \quad (12)$$

1.2 Approximate the wave equation

The innovative work was done by John Claerbout, well-known as 15° wave equation to separate the up-going and down-going waves [4].

Eliminating the source term, the Fourier transform of the scalar wave equation (Eq. (5)) can be specified as:

$$\frac{\omega^2}{v^2} = k_x^2 + k_z^2. \quad (13)$$

The down-going wave equation in Fourier domain is

$$k_z = \sqrt{\frac{\omega^2}{v^2} - k_x^2} = \frac{\omega}{v} \sqrt{1 - \frac{v^2 k_x^2}{\omega^2}}. \quad (14)$$

Using the different order Pade expansions, we have:

$$\left\{ \begin{array}{l} 1st - order : k_z = \frac{\omega}{v} \left(1 - \frac{v^2 k_x^2}{2\omega^2} \right) \\ 2nd - order : k_z = \frac{\omega}{v} \left(1 - \frac{\frac{2v^2 k_x^2}{\omega^2}}{4 - \frac{v^2 k_x^2}{\omega^2}} \right) \\ 3rd - order : k_z = \frac{\omega}{v} \left(1 - \frac{\frac{v^2 k_x^2}{2\omega^2} - \frac{v^4 k_x^4}{8\omega^4}}{1 - \frac{v^2 k_x^2}{2\omega^2}} \right) \\ 4th - order : k_z = \frac{\omega}{v} \left(1 - \frac{\frac{v^2 k_x^2}{2\omega^2} - \frac{v^4 k_x^4}{4\omega^4}}{1 - \frac{3v^2 k_x^2}{4\omega^2} + \frac{v^4 k_x^4}{16\omega^4}} \right) \end{array} \right. \quad (15)$$

The corresponding time domain equations are:

$$\left\{ \begin{array}{l} 1st - order : \frac{\partial^2 u}{\partial z^2} + \frac{v}{2} \frac{\partial^2 u}{\partial x^2} - \frac{1}{v} \frac{\partial^2 u}{\partial t^2} = 0, (the well - known 15^\circ wave equation) \\ 2nd - order : \frac{\partial^3 u}{\partial t^2 \partial z} - \frac{v^2}{4} \frac{\partial^3 u}{\partial x^2 \partial z} - \frac{1}{v} \frac{\partial^3 u}{\partial t^3} + \frac{3v}{4} \frac{\partial^3 u}{\partial x^2 \partial t} = 0 \\ 3rd - order : \frac{\partial^4 u}{\partial t^3 \partial z} - \frac{v^2}{2} \frac{\partial^4 u}{\partial x^2 \partial t \partial z} - \frac{1}{v} \frac{\partial^4 u}{\partial t^4} + v \frac{\partial^4 u}{\partial x^2 \partial t^2} - \frac{v^3}{8} \frac{\partial^4 u}{\partial x^4} = 0 \\ 4th - order : \frac{\partial^5 u}{\partial t^4 \partial z} - \frac{3v^2}{4} \frac{\partial^5 u}{\partial x^2 \partial t^2 \partial z} - \frac{v^4}{16} \frac{\partial^5 u}{\partial x^4 \partial z} + \frac{1}{v} \frac{\partial^5 u}{\partial t^5} + \frac{5v}{4} \frac{\partial^5 u}{\partial x^2 \partial t^3} + \frac{5v^3}{16} \frac{\partial^5 u}{\partial t \partial x^4} = 0 \end{array} \right. \quad (16)$$

2 Forward modelling

2.1 Elastic wave equation

In elastic wave equation, the modulus $\kappa(\mathbf{x})$ corresponds to two Lamé parameters: $\lambda = \rho(v_p^2 - 2v_s^2)$ and $\mu = \rho v_s^2$, in which v_p and v_s denote the p- and s-wave velocity. The elastic wave equation can be written as

$$\left\{ \begin{array}{l} \frac{\partial v_x}{\partial t} = \frac{1}{\rho} \left(\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial x} \right) \\ \frac{\partial v_z}{\partial t} = \frac{1}{\rho} \left(\frac{\partial \tau_{zx}}{\partial z} + \frac{\partial \tau_{zz}}{\partial z} \right) \\ \frac{\partial \tau_{xx}}{\partial t} = c_{11} \frac{\partial v_x}{\partial x} + c_{13} \frac{\partial v_z}{\partial z} \\ \frac{\partial \tau_{zz}}{\partial t} = c_{13} \frac{\partial v_x}{\partial x} + c_{33} \frac{\partial v_z}{\partial z} \\ \frac{\partial \tau_{xz}}{\partial t} = c_{44} \frac{\partial v_x}{\partial x} + c_{44} \frac{\partial v_z}{\partial z} \end{array} \right. \quad (17)$$

where τ_{ij} (sometimes σ_{ij}) is stress, v_i is velocity, $i, j = x, z$. c_{ij} are material parameters. When the medium is isotropic, $c_{11} = c_{33} = \lambda + 2\mu$, $c_{13} = \lambda$, $c_{44} = \mu$.

2.2 Absorbing boundary condition (ABC)

To simulate the wave propagation in the infinite space, the absorbing boundary condition (ABC), namely the proximal approximation (PA) boundary condition, was proposed in [6]. The basic idea is to use the

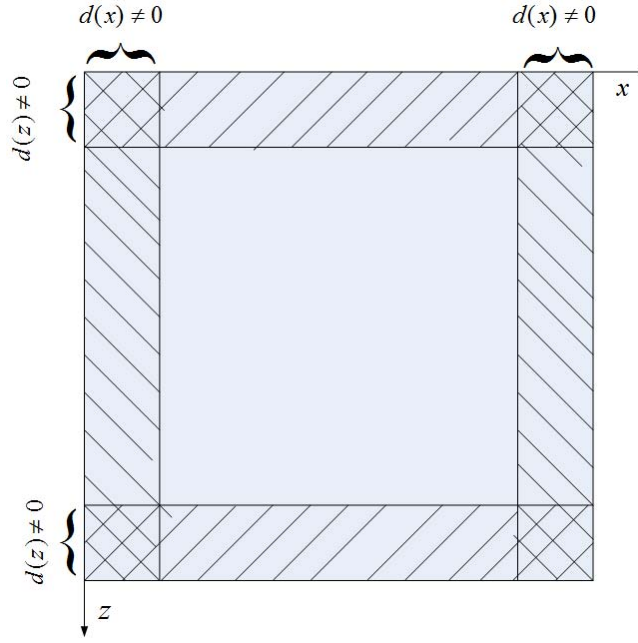


Figure 1: A schematic diagram of PML zone

wave equation with opposite direction at the boundary. Take the bottom boundary as an example. Here, allowing for the incident wave is down-going, we use the up-going wave equation at the bottom boundary. A 2nd-order PA ABC for forward modelling using CUAD programming is shown in Appendix A.

2.3 Perfectly Matched Layer (PML)

The PML ABC was proposed in electromagnetics computation [2].

2.3.1 Split PML (SPML) for acoustics

The SPML governing equation can be specified as [5]

$$\begin{cases} p = p_x + p_z \\ \frac{\partial p_x}{\partial t} + d(x)p_x = v^2 \frac{\partial v_x}{\partial x} \\ \frac{\partial p_z}{\partial t} + d(z)p_z = v^2 \frac{\partial v_z}{\partial z} \\ \frac{\partial v_x}{\partial t} + d(x)v_x = \frac{\partial p}{\partial x} \\ \frac{\partial v_z}{\partial t} + d(z)v_z = \frac{\partial p}{\partial z} \end{cases} \quad (18)$$

where $d(x)$ and $d(z)$ are the ABC coefficients designed to attenuate the reflection in the boundary zone, see Fig. 1. There exists many forms of ABC coefficients function. In the absorbing layers, we use the following model for the damping parameter $d(x)$ [5]:

$$d(x) = d_0 \left(\frac{x}{L}\right)^2, d_0 = -\frac{3v}{2L} \ln(R) \quad (19)$$

where L indicates the PML thickness; x represents the distance between current position (in PML) and PML inner boundary. R is always chosen as $10^{-3} \sim 10^{-6}$. It is also important to note that the same idea can be applied to elastic wave equation.

2.3.2 Nonsplit Convolutional-PML (CPML) for acoustics

Another approach to improve the behavior of the discrete PML at grazing incidence consists in modifying the complex coordinate transform used classically in the PML to introduce a frequency-dependent term that implements a Butterworth-type filter in the layer. This approach has been developed for Maxwell's equations named convolutional PML (CPML) [12] or complex frequency shifted-PML (CFS-PML). The

key idea is that for waves whose incidence is close to normal, the presence of such a filter changes almost nothing because absorption is already almost perfect. But for waves with grazing incidence, which for geometrical reasons do not penetrate very deep in the PML, but travel there a longer way in the direction parallel to the layer, adding such a filter will strongly attenuate them and will prevent them from leaving the PML with significant energy .

Define $Ax = \frac{\partial p}{\partial x}$, $Az = \frac{\partial p}{\partial z}$. Then the acoustic wave equation reads

$$\frac{\partial^2 p}{\partial t^2} = v^2 \left(\frac{\partial Ax}{\partial x} + \frac{\partial Az}{\partial z} \right).$$

To combine the absorbing effects into the acoustic equation, we merely need to combine two convolution terms into the above equations:

$$\begin{cases} \frac{\partial^2 p}{\partial t^2} = v^2 (Px + Pz) \\ Px = \frac{\partial Ax}{\partial x} + \Psi_x \\ Pz = \frac{\partial Az}{\partial z} + \Psi_z \\ Ax = \frac{\partial p}{\partial x} + \Phi_x \\ Az = \frac{\partial p}{\partial z} + \Phi_z \end{cases} \quad (20)$$

where Ψ_x , Ψ_z are the convolution terms of Ax and Az ; Φ_x , Φ_z are the convolution terms of Px and Pz . These convolution terms can be computed via the following relation:

$$\begin{cases} \Psi_x^n = b_x \Psi_x^{n-1} + (b_x - 1) \partial_x^{n+1/2} Ax \\ \Psi_z^n = b_z \Psi_z^{n-1} + (b_z - 1) \partial_z^{n+1/2} Az \\ \Phi_x^n = b_x \Phi_x^{n-1} + (b_x - 1) \partial_x^{n+1/2} Px \\ \Phi_z^n = b_z \Phi_z^{n-1} + (b_z - 1) \partial_z^{n+1/2} Pz \end{cases} \quad (21)$$

where $b_x = e^{-d(x)\Delta t}$ and $b_z = e^{-d(z)\Delta t}$. More details about the derivation of C-PML, the interested readers are referred to [5] and [9].

2.3.3 Non-split PML (NPML) for elastics

The non-split elastic wave equation is

$$\begin{cases} \rho \frac{\partial v_x}{\partial t} = \left(\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z} \right) - \Omega_{xx} - \Omega_{xz} \\ \rho \frac{\partial v_z}{\partial t} = \left(\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z} \right) - \Omega_{zx} - \Omega_{zz} \\ \frac{\partial \tau_{xx}}{\partial t} = (\lambda + 2\mu) \frac{\partial v_x}{\partial x} + \lambda \frac{\partial v_z}{\partial z} - (\lambda + 2\mu) \Psi_{xx} - \lambda \Psi_{zz} \\ \frac{\partial \tau_{zz}}{\partial t} = \lambda \frac{\partial v_x}{\partial x} + (\lambda + 2\mu) \frac{\partial v_z}{\partial z} - \lambda \Psi_{xx} - (\lambda + 2\mu) \Psi_{zz} \\ \frac{\partial \tau_{xz}}{\partial t} = \mu \frac{\partial v_x}{\partial z} + \mu \frac{\partial v_z}{\partial x} - \mu \Psi_{zx} - \mu \Psi_{xz} \end{cases} \quad (22)$$

where the auxiliary variables are governed via the following relation

$$\begin{cases} \frac{\partial \Omega_{xx}}{\partial t} + d(x) \Omega_{xx} = d(x) \frac{\partial \tau_{xx}}{\partial x}, \frac{\partial \Omega_{xz}}{\partial t} + d(z) \Omega_{xz} = d(z) \frac{\partial \tau_{xz}}{\partial z} \\ \frac{\partial \Omega_{zx}}{\partial t} + d(x) \Omega_{zx} = d(x) \frac{\partial \tau_{zx}}{\partial x}, \frac{\partial \Omega_{zz}}{\partial t} + d(z) \Omega_{zz} = d(z) \frac{\partial \tau_{zz}}{\partial z} \\ \frac{\partial \Psi_{xx}}{\partial t} + d(x) \Psi_{xx} = d(x) \frac{\partial v_x}{\partial x}, \frac{\partial \Psi_{xz}}{\partial t} + d(z) \Psi_{xz} = d(z) \frac{\partial v_x}{\partial z} \\ \frac{\partial \Psi_{zx}}{\partial t} + d(x) \Psi_{zx} = d(x) \frac{\partial v_z}{\partial x}, \frac{\partial \Psi_{zz}}{\partial t} + d(z) \Psi_{zz} = d(z) \frac{\partial v_z}{\partial z} \end{cases} \quad (23)$$

2.4 Discretization

The Taylor series expansion of a function $f(x)$ can be written as

$$\begin{cases} f(x+h) = f(x) + \frac{\partial f(x)}{\partial x}h + \frac{1}{2!}\frac{\partial^2 f(x)}{\partial x^2}h^2 + \frac{1}{3!}\frac{\partial^3 f(x)}{\partial x^3}h^3 + \dots \\ f(x-h) = f(x) - \frac{\partial f(x)}{\partial x}h + \frac{1}{2!}\frac{\partial^2 f(x)}{\partial x^2}h^2 - \frac{1}{3!}\frac{\partial^3 f(x)}{\partial x^3}h^3 + \dots \end{cases} \quad (24)$$

It leads to

$$\begin{cases} \frac{f(x+h) + f(x-h)}{2} = f(x) + \frac{1}{2!}\frac{\partial^2 f(x)}{\partial x^2}h^2 + \frac{1}{4!}\frac{\partial^4 f(x)}{\partial x^4}h^4 + \dots \\ \frac{f(x+h) - f(x-h)}{2} = \frac{\partial f(x)}{\partial x}h + \frac{1}{3!}\frac{\partial^3 f(x)}{\partial x^3}h^3 + \frac{1}{5!}\frac{\partial^5 f(x)}{\partial x^5}h^5 + \dots \end{cases} \quad (25)$$

Let $h = \Delta x/2$. This implies

$$\begin{cases} \frac{\partial f(x)}{\partial x} = \frac{f(x+\Delta x/2) - f(x-\Delta x/2)}{\Delta x} + O(\Delta x^2) \\ f(x) = \frac{f(x+\Delta x/2) + f(x-\Delta x/2)}{2} + O(\Delta x^2) \end{cases} \quad (26)$$

2.4.1 Higher-order approximation of staggered-grid finite difference

To approximate the 1st-order derivatives as accurate as possible, we express it in the following

$$\begin{aligned} \frac{\partial f}{\partial x} &= a_1 \frac{f(x+\Delta x/2) - f(x-\Delta x/2)}{\Delta x} + a_2 \frac{f(x+3\Delta x/2) - f(x-3\Delta x/2)}{3\Delta x} + a_3 \frac{f(x+5\Delta x/2) - f(x-5\Delta x/2)}{5\Delta x} + \dots \\ &= c_1 \frac{f(x+\Delta x/2) - f(x-\Delta x/2)}{\Delta x} + c_2 \frac{f(x+3\Delta x/2) - f(x-3\Delta x/2)}{\Delta x} + c_3 \frac{f(x+5\Delta x/2) - f(x-5\Delta x/2)}{\Delta x} + \dots \end{aligned} \quad (27)$$

where $c_i = a_i/(2i-1)$. Substituting the $f(x+h)$ and $f(x-h)$ with (24) for $h = \Delta x/2, 3\Delta x/2, \dots$ results in

$$\begin{aligned} \frac{\partial f}{\partial x} &= c_1 \left(\Delta x \frac{\partial f}{\partial x} + \frac{1}{3} \left(\frac{\Delta x}{2} \right)^2 \frac{\partial^3 f}{\partial x^3} + \dots \right) / \Delta x \\ &\quad + c_2 \left(3\Delta x \frac{\partial f}{\partial x} + \frac{1}{3} \left(\frac{3\Delta x}{2} \right)^2 \frac{\partial^3 f}{\partial x^3} + \dots \right) / \Delta x \\ &\quad + c_3 \left(5\Delta x \frac{\partial f}{\partial x} + \frac{1}{3} \left(\frac{5\Delta x}{2} \right)^2 \frac{\partial^3 f}{\partial x^3} + \dots \right) / \Delta x + \dots \\ &= (c_1 + 3c_2 + 5c_3 + 7c_4 + \dots) \frac{\partial f}{\partial x} \\ &\quad + \frac{\Delta x^2}{3 \cdot 2^2} (c_1 + 3^3 c_2 + 5^3 c_3 + 7^3 c_4 + \dots) \frac{\partial^3 f}{\partial x^3} \\ &\quad + \frac{\Delta x^4}{3 \cdot 2^4} (c_1 + 3^5 c_2 + 5^5 c_3 + 7^5 c_4 + \dots) \frac{\partial^5 f}{\partial x^5} + \dots \\ &= (a_1 + a_2 + a_3 + a_4 + \dots) \frac{\partial f}{\partial x} \\ &\quad + \frac{\Delta x^2}{3 \cdot 2^2} (a_1 + 3^2 a_2 + 5^2 a_3 + 7^2 a_4 + \dots) \frac{\partial^3 f}{\partial x^3} \\ &\quad + \frac{\Delta x^4}{3 \cdot 2^4} (a_1 + 3^4 a_2 + 5^4 a_3 + 7^4 a_4 + \dots) \frac{\partial^5 f}{\partial x^5} + \dots \end{aligned} \quad (28)$$

Thus, taking first N terms means

$$\begin{cases} a_1 + a_2 + a_3 + \dots + a_N = 1 \\ a_1 + 3^2 a_2 + 5^2 a_3 + \dots + (2N-1)^2 a_N = 0 \\ a_1 + 3^4 a_2 + 5^4 a_3 + \dots + (2N-1)^4 a_N = 0 \\ \dots \\ a_1 + 3^{2N-2} a_2 + 5^{2N-2} a_3 + \dots + (2N-1)^{2N-2} a_N = 0 \end{cases} \quad (29)$$

In matrix form,

$$\underbrace{\begin{bmatrix} 1 & 1 & \dots & 1 \\ 1^2 & 3^2 & \dots & (2N-1)^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1^{2N-2} & 3^{2N-2} & \dots & (2N-1)^{2N-2} \end{bmatrix}}_{\mathbf{V}^T} \underbrace{\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\mathbf{b}} \quad (30)$$

The above matrix equation is Vandermonde-like system: $\mathbf{V}^T \mathbf{a} = \mathbf{b}$, $\mathbf{a} = (a_1, a_2, \dots, a_N)^T$. The Vandermonde matrix

$$\mathbf{V}^T = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_N \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{N-1} & x_2^{N-1} & \dots & x_N^{N-1} \end{bmatrix} \quad (31)$$

in which $x_i = (2i-1)^2$, has analytic solutions. $\mathbf{V}^T \mathbf{a} = \mathbf{b}$ can be solved using the specific algorithms, see [3]. And we obtain

$$\frac{\partial f}{\partial x} = \frac{1}{\Delta x} \sum_{i=1}^N c_i (f(x + i\Delta x/2) - f(x - i\Delta x/2)) + O(\Delta x^{2N}) \quad (32)$$

The MATLAB code for solving the $2N$ -order finite difference coefficients is provided in Appendix B. In general, the stability of staggered-grid difference requires that

$$\Delta t \max(v) \sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta z^2}} \leq \frac{1}{\sum_{i=1}^N |c_i|}. \quad (33)$$

Define $C = \frac{1}{\sum_{i=1}^N |c_i|}$. Then, we have

$$\begin{cases} N = 1, & C = 1 \\ N = 2, & C = 0.8571 \\ N = 3, & C = 0.8054 \\ N = 4, & C = 0.7774 \\ N = 5, & C = 0.7595 \end{cases}$$

In the 2nd-order case, numerical dispersion is limited when

$$\max(\Delta x, \Delta z) < \frac{\min(v)}{10f_{\max}}. \quad (34)$$

The 4th-order dispersion relation is:

$$\max(\Delta x, \Delta z) < \frac{\min(v)}{5f_{\max}}. \quad (35)$$

2.4.2 Discretization of SPML

Take

$$\frac{\partial p_x}{\partial t} + d(x)p_x = v^2 \frac{\partial v_x}{\partial x}$$

for an example. Using the 2nd-order approximation in Eq. (26), we expand it at the time $(k+1/2)\Delta t$ and the point $[ix\Delta x, iz\Delta z]$

$$\frac{p_x^{k+1}[ix, iz] - p_x^k[ix, iz]}{\Delta t} + d[ix] \frac{p_x^{k+1}[ix, iz] + p_x^k[ix, iz]}{2} = v^2[ix, iz] \frac{v_x^{k+1/2}[ix+1/2, iz] - v_x^{k+1/2}[ix-1/2, iz]}{\Delta x} \quad (36)$$

That is to say,

$$p_x^{k+1}[ix, iz] = \frac{1 - 0.5\Delta t d[ix]}{1 + 0.5\Delta t d[ix]} p_x^k[ix, iz] + \frac{1}{1 + 0.5\Delta t d[ix]} \frac{\Delta t v^2[ix, iz]}{\Delta x} (v_x^{k+1/2}[ix+1/2, iz] - v_x^{k+1/2}[ix-1/2, iz]) \quad (37)$$

in which $d[ix] = d[ix, iz], \forall iz \in PML$. At time $k\Delta t$ and $[ix+1/2, iz]$, we expand

$$\frac{\partial v_x}{\partial t} + d(x)v_x = \frac{\partial p}{\partial x}$$

as

$$\frac{v_x^{k+1/2}[ix + 1/2, iz] - v_x^{k-1/2}[ix + 1/2, iz]}{\Delta t} + d[ix] \frac{v_x^{k+1/2}[ix + 1/2, iz] + v_x^{k-1/2}[ix, iz]}{2} = \frac{p^k[ix + 1, iz] - p^k[ix, iz]}{\Delta x} \quad (38)$$

Thus, we have

$$v_x^{k+1/2}[ix + 1/2, iz] = \frac{1 - 0.5\Delta t d[ix]}{1 + 0.5\Delta t d[ix]} v_x^{k-1/2}[ix + 1/2, iz] + \frac{1}{1 + 0.5\Delta t d[ix]} \frac{\Delta t v^2[ix, iz]}{\Delta x} (p^k[ix + 1, iz] - p^k[ix, iz]) \quad (39)$$

In summary,

$$\left\{ \begin{array}{l} v_x^{k+1/2}[ix + 1/2, iz] = \frac{1 - 0.5\Delta t d[ix]}{1 + 0.5\Delta t d[ix]} v_x^{k-1/2}[ix + 1/2, iz] + \frac{1}{1 + 0.5\Delta t d[ix]} \frac{\Delta t v^2[ix, iz]}{\Delta x} (p^k[ix + 1, iz] - p^k[ix, iz]) \\ v_z^{k+1/2}[ix, iz + 1/2] = \frac{1 - 0.5\Delta t d[iz]}{1 + 0.5\Delta t d[iz]} v_z^{k-1/2}[ix, iz + 1/2] + \frac{1}{1 + 0.5\Delta t d[iz]} \frac{\Delta t v^2[ix, iz]}{\Delta z} (p^k[ix, iz + 1] - p^k[ix, iz]) \\ p_x^{k+1}[ix, iz] = \frac{1 - 0.5\Delta t d[ix]}{1 + 0.5\Delta t d[ix]} p_x^k[ix, iz] + \frac{1}{1 + 0.5\Delta t d[ix]} \frac{\Delta t v^2[ix, iz]}{\Delta x} (v_x^{k+1/2}[ix + 1/2, iz] - v_x^{k+1/2}[ix - 1/2, iz]) \\ p_z^{k+1}[ix, iz] = \frac{1 - 0.5\Delta t d[iz]}{1 + 0.5\Delta t d[iz]} p_z^k[ix, iz] + \frac{1}{1 + 0.5\Delta t d[iz]} \frac{\Delta t v^2[ix, iz]}{\Delta z} (v_z^{k+1/2}[ix, iz + 1/2] - v_z^{k+1/2}[ix, iz - 1/2]) \\ p^{k+1}[ix, iz] = p_x^{k+1}[ix, iz] + p_z^{k+1}[ix, iz] \end{array} \right. \quad (40)$$

If we define:

$$b' = \frac{1 - 0.5\Delta t d}{1 + 0.5\Delta t d}, b = \exp(-\Delta t d) \quad (41)$$

we can easily find that b' is a good approximation of b up to 2nd order, allowing for the 2nd order Pade expansion:

$$\exp(z) \approx \frac{1 + 0.5z}{1 - 0.5z} \quad (42)$$

Then, we have

$$1 - b \approx 1 - b' = \frac{\Delta t d}{1 + 0.5\Delta t d} \quad (43)$$

2.4.3 Discretization of NPML

Note that all sub-equations can be formulated in the following form:

$$\frac{\partial f}{\partial t} + df = \gamma. \quad (44)$$

The analytic solution of this equation is

$$f = -\frac{1}{d} e^{-dt} + \frac{1}{d} \gamma \quad (45)$$

In discrete form,

$$\begin{aligned} f((k)\Delta t) &= -\frac{1}{d} e^{-dk\Delta t} + \frac{1}{d} \gamma, \\ f((k+1)\Delta t) &= -\frac{1}{d} e^{-d(k+1)\Delta t} + \frac{1}{d} \gamma. \end{aligned} \quad (46)$$

Thus,

$$f((k+1)\Delta t) = e^{-d\Delta t} f(k\Delta t) + \frac{1}{d} (1 - e^{-d\Delta t}) \gamma \quad (47)$$

For $\frac{\partial \Omega_{xx}}{\partial t} + d(x)\Omega_{xx} = d(x)\frac{\partial \tau_{xx}}{\partial x}$, $\gamma = d(x)\frac{\partial \tau_{xx}}{\partial x}$, the update rule becomes

$$\Omega_{xx}^{k+1} = e^{-d(x)\Delta t} \Omega_{xx}^k + (1 - e^{-d(x)\Delta t}) \frac{\partial \tau_{xx}^{k+1/2}}{\partial x} = b_x \Omega_{xx}^k + (1 - b_x) \frac{\partial \tau_{xx}^{k+1/2}}{\partial x} \quad (48)$$

where $b_x = e^{-d(x)\Delta t}$ and $b_z = e^{-d(z)\Delta t}$. Ω_{xx} , Ω_{xz} , Ω_{zx} , Ω_{zz} , Ψ_{xx} , Ψ_{xz} , Ψ_{zx} and Ψ_{zz} can be obtained in the same way:

$$\begin{cases} \Omega_{xx}^{k+1} = b_x \Omega_{xx}^k + (1 - b_x) \frac{\partial \tau_{xx}^{k+1/2}}{\partial x}, & \Omega_{xz}^{k+1} = b_z \Omega_{xz}^k + (1 - b_z) \frac{\partial \tau_{xz}^{k+1/2}}{\partial z} \\ \Omega_{zx}^{k+1} = b_x \Omega_{zx}^k + (1 - b_z) \frac{\partial \tau_{zx}^{k+1/2}}{\partial x}, & \Omega_{zz}^{k+1} = b_z \Omega_{zz}^k + (1 - b_z) \frac{\partial \tau_{zz}^{k+1/2}}{\partial z} \\ \Psi_{xx}^{k+1} = b_x \Psi_{xx}^k + (1 - b_x) \frac{\partial v_x^{k+1/2}}{\partial x}, & \Psi_{xz}^{k+1} = b_z \Psi_{xz}^k + (1 - b_z) \frac{\partial v_x^{k+1/2}}{\partial z} \\ \Psi_{zx}^{k+1} = b_x \Psi_{zx}^k + (1 - b_x) \frac{\partial v_z^{k+1/2}}{\partial x}, & \Psi_{zz}^{k+1} = b_z \Psi_{zz}^k + (1 - b_z) \frac{\partial v_z^{k+1/2}}{\partial z} \end{cases} \quad (49)$$

For acoustics, we have

$$\begin{cases} \Omega_{xx}^{k+1} = b_x \Omega_{xx}^k + (1 - b_x) \frac{\partial p^{k+1/2}}{\partial x}, \\ \Omega_{zz}^{k+1} = b_z \Omega_{zz}^k + (1 - b_z) \frac{\partial p^{k+1/2}}{\partial z}, \\ \Psi_{xx}^{k+1} = b_x \Psi_{xx}^k + (1 - b_x) \frac{\partial v_x^{k+1/2}}{\partial x}, \\ \Psi_{zz}^{k+1} = b_z \Psi_{zz}^k + (1 - b_z) \frac{\partial v_z^{k+1/2}}{\partial z}. \end{cases} \quad (50)$$

As can be seen from Eq. (22), we only need to subtract the reflection part Ω and Ψ after global updating (Eq. (17)). We summarize this procedure as follows:

step 1: perform

$$\begin{cases} v_x^{k+1/2} = v_x^{k-1/2} + \Delta t \frac{\partial p^k}{\partial x} \\ v_z^{k+1/2} = v_z^{k-1/2} + \Delta t \frac{\partial p^k}{\partial z} \\ p^{k+1} = p^k + \Delta t v^2 \left(\frac{\partial v_x^{k+1/2}}{\partial x} + \frac{\partial v_z^{k+1/2}}{\partial z} \right) \end{cases} \quad (51)$$

Step 2: In PML zone, subtract decaying parts:

$$\begin{cases} v_x^{k+1/2} = v_x^{k+1/2} - \Delta t \Psi_{xx}^k \\ v_z^{k+1/2} = v_z^{k+1/2} - \Delta t \Psi_{zz}^k \\ p_x^{k+1} = p_x^{k+1} - \Delta t v^2 \Omega_{xx}^k \\ p_z^{k+1} = p_z^{k+1} - \Delta t v^2 \Omega_{zz}^k \\ p^{k+1} = p_x^{k+1} + p_z^{k+1} \end{cases} \quad (52)$$

Define

$$M_{xx} = \Delta t v^2 \Omega_{xx}, M_{zz} = \Delta t v^2 \Omega_{zz}, W_{xx} = \Delta t \Psi_{xx}, W_{zz} = \Delta t \Psi_{zz}.$$

Then, the equations become

$$\begin{cases} v_x^{k+1/2} = v_x^{k+1/2} - W_{xx}^k \\ v_z^{k+1/2} = v_z^{k+1/2} - W_{zz}^k \\ p_x^{k+1} = p_x^{k+1} - M_{xx}^k \\ p_z^{k+1} = p_z^{k+1} - M_{zz}^k \\ p^{k+1} = p_x^{k+1} + p_z^{k+1} \end{cases} \quad (53)$$

and

$$\begin{cases} W_{xx}^{k+1} = b_x W_{xx}^k + (1 - b_x) \Delta t v^2 \frac{\partial v_x^k}{\partial x} \\ W_{zz}^{k+1} = b_z W_{zz}^k + (1 - b_z) \Delta t v^2 \frac{\partial v_z^k}{\partial z} \\ M_{xx}^{k+1} = b_x M_{xx}^k + (1 - b_x) \Delta t v^2 \frac{\partial p^{k+1/2}}{\partial x} \\ M_{zz}^{k+1} = b_z M_{zz}^k + (1 - b_z) \Delta t v^2 \frac{\partial p^{k+1/2}}{\partial z} \end{cases} \quad (54)$$

Remark: the variable p_x and p_z are only necessary in PML zone, for the convenience of memory requirement in computation.

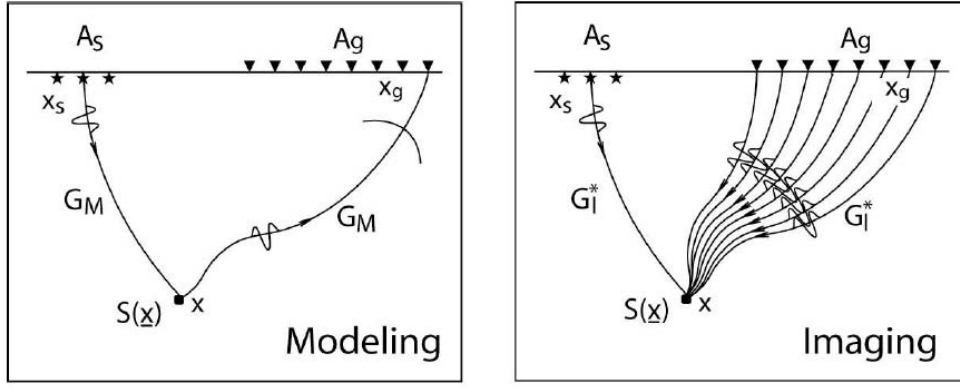


Figure 2: Modeling and migration (imaging)

3 Reverse time migration (RTM)

3.1 Features and benefits of RTM

Conventional wave equation migration is performed by propagating data downward through a velocity model into the earth and is limited where the structure and velocity field generate more complex arrivals, such as turning and 'prism' waves. Complex propagation paths give rise to arrivals that are seen as noise in the imaged data.

RTM was proposed in [1] very early. Due to the tremendous advances in computer capability, RTM has attracted more interests recently. RTM propagates events both downward and upward through the earth model, explicitly handling turning waves and all other complex propagation paths. In many cases the ability to make use of these complex wave modes allows imaging of parts of the subsurface that otherwise have poor direct illumination.

1. Features

- (a) Imaging of all possible arrivals
 - i. Superior multi-path imaging
 - ii. No dip limitation
 - iii. Accounts for extreme lateral velocity variations
- (b) Wide azimuth and TTI capable
- (c) Mirror migration of ocean bottom or VSP data
- (d) Correct for amplitude anomalies using Q-compensating RTM.

2. Benefits

- (a) Improved imaging of complex plays
 - i. Steep dips
 - ii. Complex overburdens, regardless of dip or rugosity
- (b) More accurate focusing, positioning and amplitudes in complex areas
- (c) Inclusion of TTI produces high-fidelity velocity models

3.2 RTM implementation

RTM can be carried out as follows: (1) forward-extrapolating the source wavefield, (2) backward-extrapolating the receiver wavefield, both explicitly in time, and (3) apply an imaging condition, see Fig. 2. My CUDA implementation can be precisely found in C.

3.3 Imaging condition

The cross-correlation imaging condition can be expressed as

$$I(\mathbf{x}) = \sum_{s=1}^{ns} \int_0^{t_{\max}} dt \sum_{g=1}^{ng} p_s(\mathbf{x}, t; \mathbf{x}_s) p_g(\mathbf{x}, t; \mathbf{x}_g) \quad (55)$$

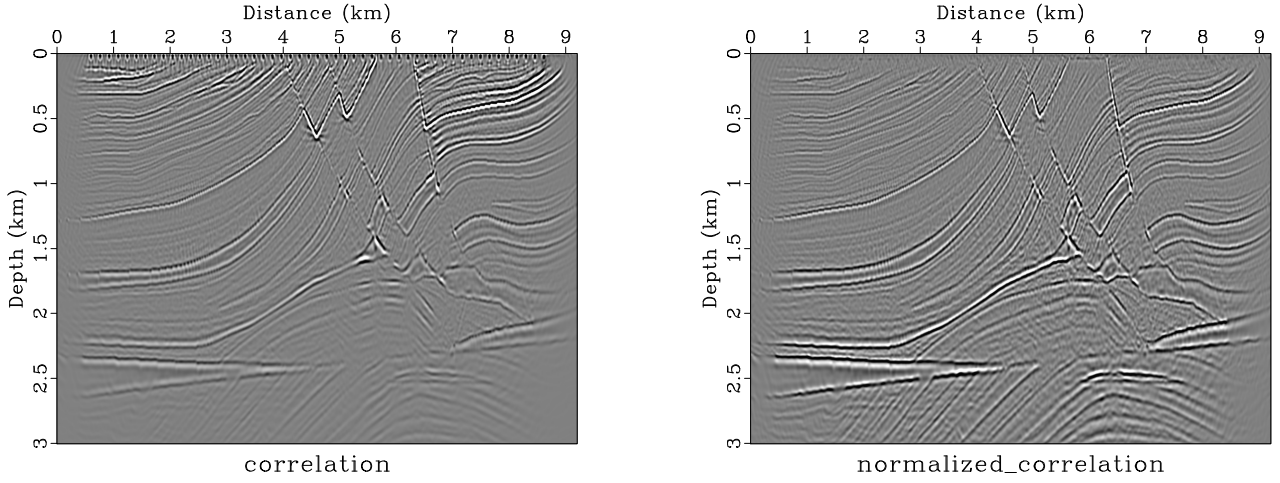


Figure 3: correlation imaging condition with and without normalization

where $I(\mathbf{x})$ is the migration image value at point \mathbf{x} ; and $p_s(\mathbf{x}, t)$ and $p_g(\mathbf{x}, t)$ are the forward and reverse-time wavefields at point \mathbf{x} . With illumination compensation, the cross-correlation imaging condition is given by

$$I(\mathbf{x}) = \sum_{s=1}^{ns} \frac{\int_0^{t_{\max}} dt \sum_{g=1}^{ng} p_s(\mathbf{x}, t; \mathbf{x}_s) p_g(\mathbf{x}, t; \mathbf{x}_g)}{\int_0^{t_{\max}} dt p_s(\mathbf{x}, t; \mathbf{x}_s) p_s(\mathbf{x}, t; \mathbf{x}_s) + \sigma^2} \quad (56)$$

in which σ^2 is chosen small to avoid being divided by zeros. I show my RTM result of the Marmousi benchmark model correlation imaging condition with and without normalization in Figure 3.

There exist a better way to carry out the illumination compensation as suggested in [7]

$$I(\mathbf{x}) = \sum_{s=1}^{ns} \frac{\int_0^{t_{\max}} dt \sum_{g=1}^{ng} p_s(\mathbf{x}, t; \mathbf{x}_s) p_g(\mathbf{x}, t; \mathbf{x}_g)}{\langle \int_0^{t_{\max}} dt p_s(\mathbf{x}, t; \mathbf{x}_s) p_s(\mathbf{x}, t; \mathbf{x}_s) \rangle_{x,y,z}} \quad (57)$$

where $\langle \rangle_{x,y,z}$ stands for smoothing in the image space in the x , y , and z directions.

Yoon and Marfurt (2006) define the seismic Poynting vector as

$$\mathbf{S} = \mathbf{v}p = \nabla p \frac{dp}{dt} = (v_x p, v_z p). \quad (58)$$

Here, we denote S_s and S_r as the source wavefield and receiver wavefield Poynting vector. Thus, the angle between the incident wave and the reflected wave can be obtained:

$$\gamma = \arccos \frac{\mathbf{S}_s \cdot \mathbf{S}_r}{|\mathbf{S}_s| |\mathbf{S}_r|} \quad (59)$$

The incident angle (or reflective angle) is half of γ , namely,

$$\theta = \frac{\gamma}{2} = \frac{1}{2} \arccos \frac{\mathbf{S}_s \cdot \mathbf{S}_r}{|\mathbf{S}_s| |\mathbf{S}_r|} \quad (60)$$

Using Poynting vector to confine the spurious artefacts, Yoon and Marfurt gives a hard thresholding scheme to weight the imaging condition [16]:

$$I(\mathbf{x}) = \sum_{s=1}^{ns} \frac{\int_0^{t_{\max}} dt \sum_{g=1}^{ng} p_s(\mathbf{x}, t; \mathbf{x}_s) p_g(\mathbf{x}, t; \mathbf{x}_g) W(\theta)}{\int_0^{t_{\max}} dt p_s(\mathbf{x}, t; \mathbf{x}_s) p_s(\mathbf{x}, t; \mathbf{x}_s) + \sigma^2} \quad (61)$$

where

$$W(\theta) = \begin{cases} 1 & \theta < \theta_{\max} \\ 0 & \text{otherwise} \end{cases} \quad (62)$$

Costa et al. (2009) modified the weight as

$$W(\theta) = \cos^3(\theta/2). \quad (63)$$

4 Full waveform inversion (FWI)

Time domain FWI was proposed by Tarantola in [13], and developed in [14, 10]. Later, frequency domain FWI was proposed by Pratt and Shin in [11]. Actually, many authors call it full waveform tomography. (tomography=fwi, imaging=migration) Here, we mainly follow two well-documented paper [11] and [15]. We define the misfit vector $\Delta \mathbf{p} = \mathbf{p}_{cal} - \mathbf{p}_{obs}$ by the differences at the receiver positions between the recorded seismic data \mathbf{p}_{obs} and the modelled seismic data $\mathbf{p}_{cal} = \mathbf{f}(\mathbf{m})$ for each source-receiver pair of the seismic survey. Here, in the simplest acoustic velocity inversion, \mathbf{m} corresponds to the velocity model to be determined. The objective function taking the least-squares norm of the misfit vector $\Delta \mathbf{p}$ is given by

$$\begin{aligned} E(\mathbf{m}) &= \frac{1}{2} \Delta \mathbf{p}^\dagger \Delta \mathbf{p} = \frac{1}{2} \Delta \mathbf{p}^T \Delta \mathbf{p}^* = \frac{1}{2} \|\mathbf{p}_{cal} - \mathbf{p}_{obs}\|^2 = \frac{1}{2} \|p_{cal}(\mathbf{x}_r) - p_{obs}(\mathbf{x}_r)\|^2 \\ &= \sum_{r=1}^{ng} |p_{cal}(\mathbf{x}_r) - p_{obs}(\mathbf{x}_r)|^2 = \sum_{r=1}^{ng} \sum_{s=1}^{ns} \int_0^{t_{max}} dt |p_{cal}(\mathbf{x}_r, t; \mathbf{x}_s) - p_{obs}(\mathbf{x}_r, t; \mathbf{x}_s)|^2 \end{aligned} \quad (64)$$

where ns and ng are the number of sources and geophones, \dagger denotes the adjoint operator (transpose conjugate), and $\mathbf{f}(\cdot)$ indicates the forward modelling of the wave propagation. The recorded seismic data is only a small subset of the whole wavefield, i.e., $\mathbf{x}_r \subset \mathbf{x}$, where $\mathbf{x}_r = [(\mathbf{x}_r)_r]$, $\mathbf{x} = [(\mathbf{x}_i)_i]$, $i = 1, 2, \dots, M$.

The minimum of the misfit function $E(\mathbf{m})$ is sought in the vicinity of the starting model \mathbf{m}_0 . The FWI is essentially a local optimization. In the framework of the Born approximation, we assume that the updated model \mathbf{m} of dimension M can be written as the sum of the starting model \mathbf{m}_0 plus a perturbation model $\Delta \mathbf{m}$: $\mathbf{m} = \mathbf{m}_0 + \Delta \mathbf{m}$. In the following, we assume that \mathbf{m} is real valued.

A second-order Taylor-Lagrange development of the misfit function in the vicinity of \mathbf{m}_0 gives the expression

$$E(\mathbf{m}_0 + \Delta \mathbf{m}) = E(\mathbf{m}_0) + \sum_{i=1}^M \frac{\partial E(\mathbf{m}_0)}{\partial m_i} \Delta m_i + \frac{1}{2} \sum_{i=1}^M \sum_{j=1}^M \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_i \partial m_j} \Delta m_i \Delta m_j + O(\|\Delta \mathbf{m}\|^3) \quad (65)$$

Taking the derivative with respect to the model parameter m_i results in

$$\frac{\partial E(\mathbf{m})}{\partial m_i} = \frac{\partial E(\mathbf{m}_0)}{\partial m_i} + \sum_{j=1}^M \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_j \partial m_i} \Delta m_j, i = 1, 2, \dots, M. \quad (66)$$

Briefly speaking, it is

$$\frac{\partial E(\mathbf{m})}{\partial \mathbf{m}} = \frac{\partial E(\mathbf{m}_0)}{\partial \mathbf{m}} + \frac{\partial^2 E(\mathbf{m}_0)}{\partial \mathbf{m}^2} \Delta \mathbf{m} \quad (67)$$

Thus,

$$\Delta \mathbf{m} = - \left(\frac{\partial^2 E(\mathbf{m}_0)}{\partial \mathbf{m}^2} \right)^{-1} \frac{\partial E(\mathbf{m}_0)}{\partial \mathbf{m}} = -\mathbf{H}^{-1} \nabla E_{\mathbf{m}} \quad (68)$$

where

$$\nabla E_{\mathbf{m}} = \frac{\partial E(\mathbf{m}_0)}{\partial \mathbf{m}} = \left[\frac{\partial E(\mathbf{m}_0)}{\partial m_1}, \frac{\partial E(\mathbf{m}_0)}{\partial m_2}, \dots, \frac{\partial E(\mathbf{m}_0)}{\partial m_M} \right]^T \quad (69)$$

and

$$\mathbf{H} = \frac{\partial^2 E(\mathbf{m}_0)}{\partial \mathbf{m}^2} = \left[\left(\frac{\partial^2 E(\mathbf{m}_0)}{\partial m_i \partial m_j} \right)_{i,j} \right] = \begin{bmatrix} \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_1^2} & \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_1 \partial m_2} & \cdots & \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_1 \partial m_M} \\ \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_2 \partial m_1} & \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_2^2} & \cdots & \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_2 \partial m_M} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_M \partial m_1} & \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_M \partial m_2} & \cdots & \frac{\partial^2 E(\mathbf{m}_0)}{\partial m_M^2} \end{bmatrix}. \quad (70)$$

$\nabla E_{\mathbf{m}}$ and \mathbf{H} are the gradient vector and the Hessian matrix, respectively.

4.1 The Newton, Gauss-Newton, and steepest-descent methods

Denote $p_{cal}^r = p_{cal}(\mathbf{x}_r)$ and $p_{obs}^r = p_{obs}(\mathbf{x}_r)$ for brevity. In terms of Eq. (64),

$$\begin{aligned}
\frac{\partial E(\mathbf{m})}{\partial m_i} &= \frac{1}{2} \sum_{r=1}^{ng} \left[\left(\frac{\partial p_{cal}^r}{\partial m_i} \right) (p_{cal}^r - p_{obs}^r)^* + \left(\frac{\partial p_{cal}^r}{\partial m_i} \right)^* (p_{cal}^r - p_{obs}^r) \right] \\
&= \sum_{r=1}^{ng} \text{Re} \left[\left(\frac{\partial p_{cal}^r}{\partial m_i} \right)^* (p_{cal}^r - p_{obs}^r) \right] \\
&= \sum_{r=1}^{ng} \text{Re} \left[\left(\frac{\partial p_{cal}^r}{\partial m_i} \right)^* \Delta p^r \right] (\Delta p^r = p_{cal}^r - p_{obs}^r) \\
&= \text{Re} \left\{ \left[\left(\frac{\partial p_{cal}^1}{\partial m_i} \right)^*, \left(\frac{\partial p_{cal}^2}{\partial m_i} \right)^*, \dots, \left(\frac{\partial p_{cal}^{ng}}{\partial m_i} \right)^* \right] \begin{bmatrix} \Delta p^1 \\ \Delta p^2 \\ \vdots \\ \Delta p^{ng} \end{bmatrix} \right\} \\
&= \text{Re} \left[\left(\frac{\partial p_{cal}(\mathbf{x}_r)}{\partial m_i} \right)^\dagger \Delta p(\mathbf{x}_r) \right] = \text{Re} \left[\left(\frac{\partial \mathbf{f}(\mathbf{m})}{\partial m_i} \right)^\dagger \Delta \mathbf{p} \right], i = 1, 2, \dots, M.
\end{aligned} \tag{71}$$

That is to say,

$$\nabla E_{\mathbf{m}} = \nabla E(\mathbf{m}) = \frac{\partial E(\mathbf{m})}{\partial \mathbf{m}} = \text{Re} \left[\left(\frac{\partial \mathbf{f}(\mathbf{m})}{\partial \mathbf{m}} \right)^\dagger \Delta \mathbf{p} \right] = \text{Re} [\mathbf{J}^\dagger \Delta \mathbf{p}] \tag{72}$$

where Re takes the real part, and $\mathbf{J} = \frac{\partial \mathbf{f}(\mathbf{m})}{\partial \mathbf{m}}$ is the Jacobian matrix, i.e., the sensitivity or the Fréchet derivative matrix.

Differentiation of the gradient expression (71) with respect to the model parameters gives the following expression for the Hessian \mathbf{H} :

$$\begin{aligned}
\mathbf{H}_{i,j} &= \frac{\partial^2 E(\mathbf{m})}{\partial m_i \partial m_j} = \frac{\partial E(\mathbf{m})}{\partial m_j} \left(\frac{\partial E(\mathbf{m})}{\partial m_i} \right) \\
&= \sum_{r=1}^{ng} \text{Re} \left[\frac{\partial}{\partial m_j} \left(\left(\frac{\partial p_{cal}^r}{\partial m_i} \right)^* (p_{cal}^r - p_{obs}^r) \right) \right] \\
&= \sum_{r=1}^{ng} \text{Re} \left[\frac{\partial}{\partial m_j} \left(\frac{\partial p_{cal}^r}{\partial m_i} \right) \Delta p^{r*} + \left(\frac{\partial p_{cal}^r}{\partial m_i} \right)^* \frac{\partial p_{cal}^r}{\partial m_j} \right] \\
&= \text{Re} \left\{ \left[\frac{\partial}{\partial m_j} \left(\frac{\partial p_{cal}^1}{\partial m_i} \right), \frac{\partial}{\partial m_j} \left(\frac{\partial p_{cal}^2}{\partial m_i} \right), \dots, \frac{\partial}{\partial m_j} \left(\frac{\partial p_{cal}^{ng}}{\partial m_i} \right) \right] \begin{bmatrix} \Delta p^{1*} \\ \Delta p^{2*} \\ \vdots \\ \Delta p^{ng*} \end{bmatrix} \right\} \\
&\quad + \text{Re} \left\{ \left[\left(\frac{\partial p_{cal}^1}{\partial m_i} \right)^*, \left(\frac{\partial p_{cal}^2}{\partial m_i} \right)^*, \dots, \left(\frac{\partial p_{cal}^{ng}}{\partial m_i} \right)^* \right] \begin{bmatrix} \frac{\partial p_{cal}^1}{\partial m_j} \\ \frac{\partial p_{cal}^2}{\partial m_j} \\ \vdots \\ \frac{\partial p_{cal}^{ng}}{\partial m_j} \end{bmatrix} \right\} \\
&= \text{Re} \left[\frac{\partial}{\partial m_j} \left(\frac{\partial \mathbf{p}_{cal}}{\partial m_i} \right)^T \Delta \mathbf{p}^* \right] + \text{Re} \left[\frac{\partial \mathbf{p}_{cal}^\dagger}{\partial m_i} \frac{\partial \mathbf{p}_{cal}}{\partial m_j} \right]
\end{aligned} \tag{73}$$

In matrix form

$$\mathbf{H} = \frac{\partial^2 E(\mathbf{m})}{\partial \mathbf{m}^2} = \text{Re} [\mathbf{J}^\dagger \mathbf{J}] + \text{Re} \left[\frac{\partial \mathbf{J}^T}{\partial \mathbf{m}^T} (\Delta \mathbf{p}^*, \Delta \mathbf{p}^*, \dots, \Delta \mathbf{p}^*) \right]. \tag{74}$$

Most of the time, this second-order term is neglected for nonlinear inverse problems. In the following, the remaining term in the Hessian, i.e., $\mathbf{H}_a = \text{Re}[\mathbf{J}^\dagger \mathbf{J}]$, is referred to as the approximate Hessian. It is the auto-correlation of the derivative wavefield. Eq. (68) becomes

$$\Delta \mathbf{m} = -\mathbf{H}^{-1} \nabla E_{\mathbf{m}} = -\mathbf{H}_a^{-1} \text{Re}[\mathbf{J}^\dagger \Delta \mathbf{p}]. \tag{75}$$

The method which solves equation (74) when only \mathbf{H}_a is estimated is referred to as the Gauss-Newton method. To guarantee the stability of the algorithm (avoiding the singularity), we can use $\mathbf{H} = \mathbf{H}_a + \eta \mathbf{I}$, leading to

$$\Delta \mathbf{m} = -\mathbf{H}^{-1} \nabla E_{\mathbf{m}} = -(\mathbf{H}_a + \eta \mathbf{I})^{-1} \text{Re} [\mathbf{J}^\dagger \Delta \mathbf{p}]. \quad (76)$$

Alternatively, the inverse of the Hessian in Eq. (68) can be replaced by $\mathbf{H} = \mathbf{H}_a = \mu \mathbf{I}$, leading to the gradient or steepest-descent method:

$$\Delta \mathbf{m} = -\mu^{-1} \nabla E_{\mathbf{m}} = -\alpha \nabla E_{\mathbf{m}} = -\alpha \text{Re} [\mathbf{J}^\dagger \Delta \mathbf{p}]. \quad (77)$$

where $\alpha = \mu^{-1}$.

At the k -th iteration, the misfit function can be presented using the 2nd-order Taylor-Lagrange expansion

$$E(\mathbf{m}_{k+1}) = E(\mathbf{m}_k - \alpha_k \nabla E(\mathbf{m}_k)) = E(\mathbf{m}_k) - \alpha_k \langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle + \frac{1}{2} \alpha_k^2 \langle \mathbf{H}_k \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle. \quad (78)$$

Setting $\frac{\partial E(\mathbf{m}_{k+1})}{\partial \alpha_k} = 0$ gives

$$\alpha_k = \frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle}{\langle \mathbf{H}_k \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle} \stackrel{\mathbf{H}_k := \mathbf{H}_a = \mathbf{F}_k^\dagger \mathbf{F}_k}{=} \frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle}{\langle \mathbf{J}_k \nabla E(\mathbf{m}_k), \mathbf{J}_k \nabla E(\mathbf{m}_k) \rangle} \quad (79)$$

4.2 Conjugate gradient (CG) implementation

The gradient-like method can be summarized as

$$\mathbf{m}_{k+1} = \mathbf{m}_k + \alpha_k \mathbf{d}_k. \quad (80)$$

The conjugate gradient (CG) algorithm decreases the misfit function along the conjugate gradient direction:

$$\mathbf{d}_k = \begin{cases} -\nabla E(\mathbf{m}_0), & k = 0 \\ -\nabla E(\mathbf{m}_k) + \beta_k \mathbf{d}_{k-1}, & k \geq 1 \end{cases} \quad (81)$$

There are many ways to compute β_k :

$$\left\{ \begin{array}{l} \beta_k^{HS} = \frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1}) \rangle}{\langle \mathbf{d}_{k-1}, \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1}) \rangle} \\ \beta_k^{FR} = \frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle}{\langle \nabla E(\mathbf{m}_{k-1}), \nabla E(\mathbf{m}_{k-1}) \rangle} \\ \beta_k^{PRP} = \frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1}) \rangle}{\langle \nabla E(\mathbf{m}_{k-1}), \nabla E(\mathbf{m}_{k-1}) \rangle} \\ \beta_k^{CD} = -\frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle}{\langle \mathbf{d}_{k-1}, \nabla E(\mathbf{m}_{k-1}) \rangle} \\ \beta_k^{DY} = \frac{\langle \nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) \rangle}{\langle \mathbf{d}_{k-1}, \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1}) \rangle} \end{array} \right. \quad (82)$$

In practice, we suggest to use

$$\beta_k = \max(0, \min(\beta_k^{HS}, \beta_k^{DY})). \quad (83)$$

Iterating with Eq. (80) needs to find an appropriate α_k . Here we provide two approaches to calculate α_k .

- Approach 1: Currently, the objective function is

$$E(\mathbf{m}_{k+1}) = E(\mathbf{m}_k + \alpha_k \mathbf{d}_k) = E(\mathbf{m}_k) + \alpha_k \langle \nabla E(\mathbf{m}_k), \mathbf{d}_k \rangle + \frac{1}{2} \alpha_k^2 \langle \mathbf{H}_k \mathbf{d}_k, \mathbf{d}_k \rangle. \quad (84)$$

Setting $\frac{\partial E(\mathbf{m}_{k+1})}{\partial \alpha_k} = 0$ gives

$$\alpha_k = -\frac{\langle \mathbf{d}_k, \nabla E(\mathbf{m}_k) \rangle}{\langle \mathbf{H}_k \mathbf{d}_k, \mathbf{d}_k \rangle} \stackrel{\mathbf{H}_k := \mathbf{H}_a = \mathbf{J}_k^\dagger \mathbf{J}_k}{=} -\frac{\langle \mathbf{d}_k, \nabla E(\mathbf{m}_k) \rangle}{\langle \mathbf{J}_k \mathbf{d}_k, \mathbf{J}_k \mathbf{d}_k \rangle}. \quad (85)$$

- Approach 2: Recall that

$$\mathbf{f}(\mathbf{m}_k + \alpha_k \mathbf{d}_k) = \mathbf{f}(\mathbf{m}_k) + \frac{\partial \mathbf{f}(\mathbf{m}_k)}{\partial \mathbf{m}} \mathbf{d}_k + O(\|\mathbf{d}_k\|^2) = \mathbf{f}(\mathbf{m}_k) + \alpha_k \mathbf{J}_k \mathbf{d}_k + O(\|\mathbf{d}_k\|^2). \quad (86)$$

Using the 1st-order approximation, we have

$$\begin{aligned}
E(\mathbf{m}_{k+1}) &= \frac{1}{2} \|\mathbf{f}(\mathbf{m}_k + \alpha_k \mathbf{d}_k) - \mathbf{p}_{obs}\|^2 \\
&\approx \frac{1}{2} \|\mathbf{f}(\mathbf{m}_k) + \alpha_k \mathbf{J}_k \mathbf{d}_k - \mathbf{p}_{obs}\|^2 = \frac{1}{2} \|\mathbf{f}(\mathbf{m}_k) - \mathbf{p}_{obs} + \alpha_k \mathbf{J}_k \mathbf{d}_k\|^2 \\
&= E(\mathbf{m}) + \alpha_k \langle \mathbf{J}_k \mathbf{d}_k, \mathbf{f}(\mathbf{m}_k) - \mathbf{p}_{obs} \rangle + \frac{1}{2} \alpha_k^2 \langle \mathbf{J}_k \mathbf{d}_k, \mathbf{J}_k \mathbf{d}_k \rangle.
\end{aligned} \tag{87}$$

Setting $\frac{\partial E(\mathbf{m}_{k+1})}{\partial \alpha_k} = 0$ gives

$$\alpha_k = \frac{\langle \mathbf{J}_k \mathbf{d}_k, \mathbf{p}_{obs} - \mathbf{f}(\mathbf{m}_k) \rangle}{\langle \mathbf{J}_k \mathbf{d}_k, \mathbf{J}_k \mathbf{d}_k \rangle}. \tag{88}$$

In fact, Eq. (88) can also be obtained from Eq. (85) in terms of Eq. (72): $\nabla E_{\mathbf{m}} = \mathbf{J}^\dagger \Delta \mathbf{p}$. (Here we assume all computations are dealt with real numbers.)

In terms of Eq. (86), the term $\mathbf{J}_k \mathbf{d}_k$ is computed conventionally using a 1st-order-accurate finite difference approximation of the partial derivative of \mathbf{f} :

$$\mathbf{J}_k \mathbf{d}_k = \frac{\mathbf{f}(\mathbf{m}_k + \epsilon \mathbf{d}_k) - \mathbf{f}(\mathbf{m}_k)}{\epsilon} \tag{89}$$

with a small parameter ϵ . In practice, we chose an ϵ such that

$$\max(\epsilon |\mathbf{d}_k|) \leq \frac{\max(|\mathbf{m}_k|)}{100}. \tag{90}$$

4.3 CG method for iteratively reweighted least squares (IRLS)

In Eq. (64), define $\mathbf{r} = \Delta \mathbf{p} = \mathbf{Fm} - \mathbf{p}_{obs}$ in which we denote forward modelling $\mathbf{f}(\mathbf{m}) = \mathbf{Fm}$ for brevity. It leads to minimizing the ℓ_2 -norm of the residual: $E = \|\mathbf{r}\|_2^2/2$. The CG method can be summarized in Algorithm 1:

Algorithm 1 CG method for LS solution

```

1: initialize  $\mathbf{m}, \beta \leftarrow 0, \mathbf{g}_1 \leftarrow \mathbf{0}$ 
2: for iter=0...itermax-1 do
3:    $\mathbf{g}_0 \leftarrow \mathbf{g}_1$ 
4:    $\mathbf{r} \leftarrow \mathbf{Fm} - \mathbf{p}_{obs}$ 
5:   compute objective function, if converged, break
6:    $\mathbf{g}_1 \leftarrow \mathbf{F}^\dagger \mathbf{r}$ 
7:   compute  $\beta$  according to Eq. (82)
8:    $\mathbf{cg} \leftarrow -\mathbf{g}_1 + \beta \mathbf{cg}$ 
9:   compute  $\alpha$  according to Eq. (85) or (88)
10:   $\mathbf{m} \leftarrow \mathbf{m} + \alpha \mathbf{cg}$ 
11: end for
```

As a matter of fact, ℓ_p -norm ($0 < p < 1$) minimization can also be obtained with the method of iteratively reweighted least squares (IRLS) [8], based on the fact that

$$\|\mathbf{W}_r \mathbf{r}\|_2^2 = \mathbf{r}^\dagger \mathbf{W}_r^\dagger \mathbf{W}_r \mathbf{r} = \mathbf{r}^\dagger \mathbf{W}_r^2 \mathbf{r} = \|\mathbf{r}\|_p^p \tag{91}$$

when we take

$$\mathbf{r} = \mathbf{W}_r (\mathbf{Fm} - \mathbf{p}_{obs}), \tag{92}$$

with

$$\text{diag}(\mathbf{W}_r)_i = |r_i|^{(p-2)/2}. \tag{93}$$

Here we introduce model weight as the precondition to solve the problem

$$\mathbf{FW}_m \hat{\mathbf{m}} = \mathbf{p}_{obs} \tag{94}$$

followed by

$$\mathbf{m} = \mathbf{W}_m \hat{\mathbf{m}} \tag{95}$$

with

$$\text{diag}(\mathbf{W}_m)_i = |m_i|^{(2-p)/2}. \tag{96}$$

It is easy to verify that

$$\|\hat{\mathbf{m}}\|_2^2 = \hat{\mathbf{m}}^\dagger \hat{\mathbf{m}} = \mathbf{m}^\dagger \mathbf{W}_m^{-1} \mathbf{W}_m^{-1} \mathbf{m} = \|\mathbf{m}\|_p^p. \tag{97}$$

The CG method for IRLS solution can be conducted in Algorithm 2:

Algorithm 2 CG method for IRLS solution

```

1: initialize  $\mathbf{m}$ ,  $\beta \leftarrow 0$ ,  $\mathbf{g}_1 \leftarrow \mathbf{0}$ 
2: for iter=0...itermax1-1 do
3:   compute  $\mathbf{W}_r$  according to Eq. (93)
4:   compute  $\mathbf{W}_m$  according to Eq. (96)
5:    $\mathbf{r} \leftarrow \mathbf{W}_r(\mathbf{F}\mathbf{W}_m\mathbf{m} - \mathbf{p}_{obs})$ 
6:   compute objective function, if converged, break
7:    $\mathbf{g}_0 \leftarrow \mathbf{g}_1$ 
8:    $\mathbf{g}_1 \leftarrow \mathbf{F}^T \mathbf{r}$ 
9:   compute  $\beta$  according to Eq. (82)
10:   $\mathbf{cg} \leftarrow -\mathbf{g}_1 + \beta \mathbf{cg}$ 
11:  compute  $\alpha$  according to Eq. (85) or (88)
12:   $\mathbf{m} \leftarrow \mathbf{m} + \alpha \mathbf{cg}$ 
13:   $\mathbf{m} \leftarrow \mathbf{W}_m \mathbf{m}$ 
14: end for

```

4.4 Fréchet derivative

Recall that the basic acoustic wave equation can be specified as

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} - \nabla^2 p(\mathbf{x}, t; \mathbf{x}_s) = f_s(\mathbf{x}, t; \mathbf{x}_s). \quad (98)$$

where $f_s(\mathbf{x}, t; \mathbf{x}_s) = f_s(\mathbf{x}_s, t')\delta(\mathbf{x} - \mathbf{x}_s)\delta(t - t')$. (In Section 1, we set $t' = 0$ and abbreviated $p(\mathbf{x}, t; \mathbf{x}_s) = p(\mathbf{x}, t; \mathbf{x}_s, 0)$.) The Green's function $\Gamma(\mathbf{x}, t; \mathbf{x}_s, t')$ is defined by

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 \Gamma(\mathbf{x}, t; \mathbf{x}_s, t')}{\partial t^2} - \nabla^2 \Gamma(\mathbf{x}, t; \mathbf{x}_s, t') = \delta(\mathbf{x} - \mathbf{x}_s)\delta(t - t'). \quad (99)$$

Thus the integral representation of the solution can be given by

$$\begin{aligned} p(\mathbf{x}, t; \mathbf{x}_s) &= \int_V dV(\mathbf{x}') \int dt' \Gamma(\mathbf{x}, t; \mathbf{x}_s, t') f_s(\mathbf{x}', t; \mathbf{x}_s) \\ &= \int_V dV(\mathbf{x}') \int dt' \Gamma(\mathbf{x}, t - t'; \mathbf{x}_s, 0) f_s(\mathbf{x}', t; \mathbf{x}_s) \text{ (Causality of Green function)} \\ &= \int_V dV(\mathbf{x}') \Gamma(\mathbf{x}, t; \mathbf{x}_s, 0) * f_s(\mathbf{x}', t; \mathbf{x}_s) \end{aligned} \quad (100)$$

where $*$ denotes the convolution operator.

A perturbation of bulk modulus $v(\mathbf{x}) \rightarrow v(\mathbf{x}) + \Delta v(\mathbf{x})$ will produce a field $p(\mathbf{x}, t; \mathbf{x}_s) + \Delta p(\mathbf{x}, t; \mathbf{x}_s)$ defined by

$$\frac{1}{(v(\mathbf{x}) + \Delta v(\mathbf{x}))^2} \frac{\partial^2 [p(\mathbf{x}, t; \mathbf{x}_s) + \Delta p(\mathbf{x}, t; \mathbf{x}_s)]}{\partial t^2} - \nabla^2 [p(\mathbf{x}, t; \mathbf{x}_s) + \Delta p(\mathbf{x}, t; \mathbf{x}_s)] = f_s(\mathbf{x}, t; \mathbf{x}_s) \quad (101)$$

Note that

$$\frac{1}{(v(\mathbf{x}) + \Delta v(\mathbf{x}))^2} = \frac{1}{v^2(\mathbf{x})} - \frac{2\Delta v(\mathbf{x})}{v^3(\mathbf{x})} + O(\Delta^2 v(\mathbf{x})) \quad (102)$$

Eq. (101) subtracts Eq. (98), yielding

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 \Delta p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} - \nabla^2 \Delta p(\mathbf{x}, t; \mathbf{x}_s) = \frac{\partial^2 [p(\mathbf{x}, t; \mathbf{x}_s) + \Delta p(\mathbf{x}, t; \mathbf{x}_s)]}{\partial t^2} \frac{2\Delta v(\mathbf{x})}{v^3(\mathbf{x})} \quad (103)$$

Using the Born approximation, Eq. (103) becomes

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 \Delta p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} - \nabla^2 \Delta p(\mathbf{x}, t; \mathbf{x}_s) = \frac{\partial^2 p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} \frac{2\Delta v(\mathbf{x})}{v^3(\mathbf{x})} \quad (104)$$

Again, based on integral representation, we obtain

$$\Delta p(\mathbf{x}_r, t; \mathbf{x}_s) = \int_V dV(\mathbf{x}) \Gamma(\mathbf{x}_r, t; \mathbf{x}_s, 0) * \frac{\partial^2 p(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} \frac{2\Delta v(\mathbf{x})}{v^3(\mathbf{x})} = \int_V dV(\mathbf{x}) \Gamma(\mathbf{x}_r, t; \mathbf{x}_s, 0) * \ddot{p}(\mathbf{x}, t; \mathbf{x}_s) \frac{2\Delta v(\mathbf{x})}{v^3(\mathbf{x})} \quad (105)$$

The convolution guarantees that an input f and the system impulse response function g are exchangeable. That is to say, we can use the system impulse response function g as the input, the input f as the impulse response function, leading to the same output. In the seismic modelling and acquisition process, the same seismogram can be obtained when we shot at the receiver position \mathbf{x}_r when recording the seismic data at the position \mathbf{x} . According to (105),

$$\frac{\partial p(\mathbf{x}_r, t; \mathbf{x}_s)}{\partial v(\mathbf{x})} = \int_V dV(\mathbf{x}) \Gamma(\mathbf{x}_r, t; \mathbf{x}_s, 0) * \ddot{p}(\mathbf{x}, t; \mathbf{x}_s) \frac{2}{v^3(\mathbf{x})} = \sum_{r=1}^{ng} \Gamma(\mathbf{x}, t; \mathbf{x}_s, 0) * \ddot{p}(\mathbf{x}_r, t; \mathbf{x}_s) \frac{2}{v^3(\mathbf{x}_r)} \quad (106)$$

Note that $\mathbf{m} = [v_1(\mathbf{x}), v_2(\mathbf{x}), \dots, v_M(\mathbf{x})]^T$. The Fréchet derivative operator \mathbf{J} (Jacobian matrix) is then given by

$$(\mathbf{J}\Delta\mathbf{m})(\mathbf{x}_r, t; \mathbf{x}_s) = \Delta\mathbf{p}(\mathbf{x}_r, t; \mathbf{x}_s) = \left[\left(\sum_{r=1}^{ng} \Gamma(\mathbf{x}, t; \mathbf{x}_s, 0) * \ddot{p}(\mathbf{x}_r, t; \mathbf{x}_s) \frac{2}{v^3(\mathbf{x}_r)} \right)_i \right], i = 1, 2, \dots, M. \quad (107)$$

The Fréchet derivative in finite-dimensional spaces is the usual derivative. In particular, it is represented in coordinates by the Jacobian matrix.

4.5 Gradient computation

The convolution guarantees

$$\int dt [g(t) * f(t)] h(t) = \int dt f(t) [g(-t) * h(t)].$$

Then, Eq. (71) becomes

$$\begin{aligned} \frac{\partial E(\mathbf{m})}{\partial v(\mathbf{x})} &= \text{Re} \left[\left(\frac{\partial p_{cal}(\mathbf{x}_r)}{\partial v(\mathbf{x})} \right)^* \Delta p(\mathbf{x}_r) \right] \\ &= \sum_{r=1}^{ng} \sum_{s=1}^{ns} \int_0^{t_{\max}} dt \text{Re} \left[\left(\Gamma(\mathbf{x}, t; \mathbf{x}_s, 0) * \ddot{p}_{cal}(\mathbf{x}_r, t; \mathbf{x}_s) \frac{2}{v^3(\mathbf{x}_r)} \right)^* \Delta p(\mathbf{x}_r, t; \mathbf{x}_s) \right] \\ &= \sum_{r=1}^{ng} \sum_{s=1}^{ns} \int_0^{t_{\max}} dt \text{Re} \left[\left(\Gamma(\mathbf{x}, t; \mathbf{x}_s, 0) * \frac{\partial^2 p_{cal}(\mathbf{x}_r, t; \mathbf{x}_s)}{\partial t^2} \frac{2}{v^3(\mathbf{x}_r)} \right)^* \Delta p(\mathbf{x}_r, t; \mathbf{x}_s) \right] \\ &= \sum_{r=1}^{ng} \sum_{s=1}^{ns} \int_0^{t_{\max}} dt \text{Re} \left[\left(\frac{\partial^2 p_{cal}(\mathbf{x}_r, t; \mathbf{x}_s)}{\partial t^2} \frac{2}{v^3(\mathbf{x}_r)} \right)^* (\Gamma(\mathbf{x}, -t; \mathbf{x}_s, 0) * \Delta p(\mathbf{x}_r, t; \mathbf{x}_s)) \right] \\ &= \sum_{r=1}^{ng} \sum_{s=1}^{ns} \int_0^{t_{\max}} dt \text{Re} \left[\left(\frac{\partial^2 p_{cal}(\mathbf{x}_r, t; \mathbf{x}_s)}{\partial t^2} \frac{2}{v^3(\mathbf{x}_r)} \right)^* (\Gamma(\mathbf{x}, 0; \mathbf{x}_s, t) * \Delta p(\mathbf{x}_r, t; \mathbf{x}_s)) \right] \\ &= \sum_{r=1}^{ng} \sum_{s=1}^{ns} \int_0^{t_{\max}} dt \text{Re} \left[\left(\frac{\partial^2 p_{cal}(\mathbf{x}_r, t; \mathbf{x}_s)}{\partial t^2} \frac{2}{v^3(\mathbf{x}_r)} \right)^* p_{res}(\mathbf{x}, t; \mathbf{x}_s) \right] \end{aligned} \quad (108)$$

where $p_{res}(\mathbf{x}, t; \mathbf{x}_s) = \Gamma(\mathbf{x}, 0; \mathbf{x}_s, t) * \Delta p(\mathbf{x}_r, t; \mathbf{x}_s)$, which is a reverse-time propagated wavefield produced using the residual $\Delta p(\mathbf{x}_r, t; \mathbf{x}_s)$ as the source. That is,

$$\frac{1}{v^2(\mathbf{x})} \frac{\partial^2 p_{res}(\mathbf{x}, t; \mathbf{x}_s)}{\partial t^2} - \nabla^2 p_{res}(\mathbf{x}, t; \mathbf{x}_s) = \Delta p(\mathbf{x}_r, t; \mathbf{x}_s). \quad (109)$$

A Forward modelling with PA: CUDA code

```

1 // vel[id]==vel[id]*vel[id]
2 // dtx2=(dt/dx)^2; dtz2=(dt/dz)^2
3 #define Block_Size1      16
4 #define Block_Size2      16
5 __global__ void cuda_forward(float *p0, float *p1, float *vel, float dtx2, float
    dtz2, int nx, int nz)
6 {
7     int i1=threadIdx.x+blockIdx.x*blockDim.x;
8     int i2=threadIdx.y+blockIdx.y*blockDim.y;

```

```

9      int id=i1+i2*nz;
10
11      __shared__ float s_p0[Block_Size1+2][Block_Size2+2];
12      __shared__ float s_p1[Block_Size1+2][Block_Size2+2];
13      s_p0[threadIdx.x+1][threadIdx.y+1]=p0[id];
14      s_p1[threadIdx.x+1][threadIdx.y+1]=p1[id];
15      if(threadIdx.x==0)
16      {
17          if(blockIdx.x>0)      { s_p0[threadIdx.x][threadIdx.y+1]=p0[id
18                                -1]; s_p1[threadIdx.x][threadIdx.y+1]=p1[id-1];}
19          else                  { s_p0[threadIdx.x][threadIdx.y+1]=0.0f;
20                                s_p1[threadIdx.x][threadIdx.y+1]=0.0f;}
21      }
22      if(threadIdx.x==blockDim.x-1)
23      {
24          if(blockIdx.x<gridDim.x-1) { s_p0[threadIdx.x+2][threadIdx.
25                                        y+1]=p0[id+1]; s_p1[threadIdx.x+2][threadIdx.y+1]=p1[id+1];}
26          else                  { s_p0[threadIdx.x+2][threadIdx.
27                                y+1]=0.0f; s_p1[threadIdx.x+2][threadIdx.y+1]=0.0f;}
28      }
29      if(threadIdx.y==0)
30      {
31          if(blockIdx.y>0)      { s_p0[threadIdx.x+1][threadIdx.y]=p1[id
32                                -nz]; s_p1[threadIdx.x+1][threadIdx.y]=p1[id-nz];}
33          else                  { s_p0[threadIdx.x+1][threadIdx.y]=0.0f;
34                                s_p1[threadIdx.x+1][threadIdx.y]=0.0f;}
35      }
36      if(threadIdx.y==blockDim.y-1)
37      {
38          if(blockIdx.y<gridDim.y-1) { s_p0[threadIdx.x+1][threadIdx.
39                                        y+2]=p1[id+nz]; s_p1[threadIdx.x+1][threadIdx.y+2]=p1[id+nz
40                                        l];}
41          else                  { s_p0[threadIdx.x+1][threadIdx.
42                                y+2]=0.0f; s_p1[threadIdx.x+1][threadIdx.y+2]=0.0f;}
43      }
44      __syncthreads();
45
46      float c1=vel[id]*dtx2*(s_p1[threadIdx.x+2][threadIdx.y+1]-2.0*s_p1[
47                            threadIdx.x+1][threadIdx.y+1]+s_p1[threadIdx.x][threadIdx.y+1]);
48      float c2=vel[id]*dtz2*(s_p1[threadIdx.x+1][threadIdx.y+2]-2.0*s_p1[
49                            threadIdx.x+1][threadIdx.y+1]+s_p1[threadIdx.x+1][threadIdx.y]);
50
51      if(i1==0) // left boundary
52      {
53          c1=sqrtf(vel[id]*dtx2)*(-s_p1[threadIdx.x+1][threadIdx.y+1]+s_p1[
54                                threadIdx.x+2][threadIdx.y+1]
55                                +s_p0[threadIdx.x+1][threadIdx.y+1]-s_p0[
56                                  threadIdx.x+2][threadIdx.y+1]);
57          if(i2>0 && i2<nx-1) c2=0.5*c2;
58      }
59      if(i1==nz-1) // right boundary
60      {
61          c1=sqrtf(vel[id]*dtx2)*(s_p1[threadIdx.x][threadIdx.y+1]-s_p1[
62                                threadIdx.x+1][threadIdx.y+1]
63                                -s_p0[threadIdx.x][threadIdx.y+1]+s_p0[
64                                  threadIdx.x+1][threadIdx.y+1]);
65          if(i2>0 && i2<nx-1) c2=0.5*c2;
66      }
67      if(i2==0) // top boundary
68      {
69          if(i1>0 && i1<nz-1) c1=0.5*c1;

```

```

55         c2=sqrtf(vel[id]*dtz2)*(-s_p1[threadIdx.x+1][threadIdx.y+1]+s_p1
56             [threadIdx.x+1][threadIdx.y+2]
57             +s_p0[threadIdx.x+1][threadIdx.y+1]-s_p0
58             [threadIdx.x+1][threadIdx.y+2]);
59     }
60     if(i2==nx-1) // bottom boundary
61     {
62         if(i1>0 && i1<nz-1) c1=0.5*c1;
63         c2=sqrtf(vel[id]*dtz2)*(s_p1[threadIdx.x+1][threadIdx.y]-s_p1[
64             threadIdx.x+1][threadIdx.y+1]
65             -s_p0[threadIdx.x+1][threadIdx.y]+s_p0[
66             threadIdx.x+1][threadIdx.y+1]);
67     }
68     p0[id]=2*s_p1[threadIdx.x+1][threadIdx.y+1]-s_p0[threadIdx.x+1][
69         threadIdx.y+1]+c1+c2;
70 }

```

B Compute 2N-order difference coefficients: MATLAB code

```

1 function c=FindCoeff(NJ)
2 % Input NJ means that you are using 2*N-order difference strategy
3 % Example:
4 %     format long
5 %     NJ=10;
6 %     c=FindCoeff(NJ)
7 N=NJ/2;
8 x=zeros(N,1);
9 b=zeros(N,1); b(1)=1;
10 c=b;
11 for k=1:N
12     x(k)=(2*k-1)^2;
13 end
14
15 for k=1:N-1
16     for i=N:-1:k+1
17         b(i)=b(i)-x(k)*b(i-1);
18     end
19 end
20
21 for k=N-1:-1:1
22     for i=k+1:N
23         b(i)=b(i)/(x(i)-x(i-k));
24     end
25     for i=k:N-1
26         b(i)=b(i)-b(i+1);
27     end
28 end
29 for k=1:N
30     c(k)=b(k)/(2*k-1);
31 end

```

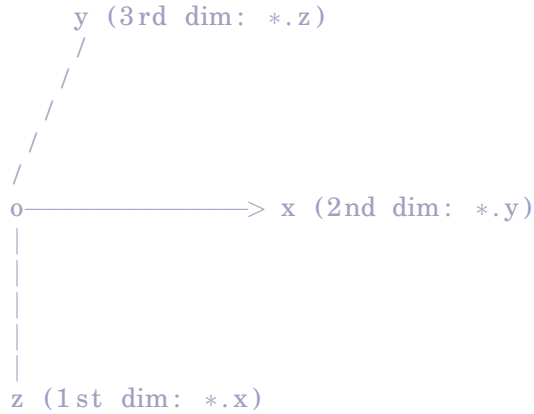
C RTM: CUDA code

```

1 /* CUDA based RTM
2 Some basic descriptions of this code are in order.

```

1) Coordinate configuration of seismic data:



```
1st dim: i1=threadIdx.x+blockDim.x*blockIdx.x;
2nd dim: i2=threadIdx.y+blockDim.y*blockIdx.y;
3rd dim: i3=threadIdx.z+blockDim.z*blockIdx.z;
(i1, i2, i3)=i1+i2*nnz+i3*nnz*nnx;
```

2) stability condition:

```
min(dx, dz)>sqrt(2)*dt*max(v) (NJ=2)
numerical dispersion condition:
max(dx, dz)<min(v)/(10*fmax) (NJ=2)
max(dx, dz)<min(v)/(5*fmax) (NJ=4)
```

3) This code doesn't save the history of forward time steps. We just save the least boundaries of every time step and the final step of the wavefield. Using this information, we can easily reconstruct the exact wavefield in the reverse time steps. It is worth noting that to implement large scale seismic imaging, pinned memory is employed to save the boundaries of each step so that all the saved data can be computed on the device directly.

4) The final images can be two kinds: result of correlation imaging condition and the normalized one. The normalized correlation imaging result is preferred due to compensated illumination. This code does not perform any kind of filtering, which is recommended if you obtained the CUDA RIM result. Some of the filters are popular and effective to remove the low frequency artifacts of the imaging: the Laplacian filtering, derivative filtering and the bandpass filtering. Personally, I prefer the bandpass filtering.

```
*/
/*
Copyright (C) 2013 Xi'an Jiaotong University (Pengliang Yang)
Email: ypl.2100@gmail.com
Acknowledgement: This code is written with the help of Baoli Wang.
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <cuda_runtime.h>

#ifdef _OPENMP
#include <omp.h>
#endif
```

```

64 #ifndef MAX
65 #define MAX(x,y) ((x) > (y) ? (x) : (y))
66 #endif
67 #ifndef MIN
68 #define MIN(x,y) ((x) < (y) ? (x) : (y))
69 #endif
70 #ifndef true
71 #define true (1)
72 #endif
73 #ifndef false
74 #define false (0)
75 #endif
76 #ifndef EPS
77 #define EPS 1.0e-15f
78 #endif
79
80 #define PI 3.141592653589793f
81 #define Block_Size1 16 // 1st dim block size
82 #define Block_Size2 16 // 2nd dim block size
83 const int npml=32;
84 const int NJ=6; // finite difference order NJ=2*k
85 const int nbell=3; // radius of Gaussian bell
86 const bool csdgather=true; // common shot gather (CSD) or not
87
88 #include "cuda_kernels.cu"
89 /*
90 const bool csdgather=false; // common shot gather (CSD) or not
91 char *model_file="sm_segsalt210x675.bin";
92 const int nz1=210;
93 const int nx1=675;
94 const float dx=20.0;
95 const float dz=20.0;
96 const float fm=30.0;
97 const float dt=0.002;
98 const int nt=3200;
99 const int ns=30;
100 const int ng=675;
101
102 int jsx=20; //source x-axis jump interval
103 int jsz=0; //source z-axis jump interval
104 int jgx=1; //geophone x-axis jump interval
105 int jgz=0; //geophone z-axis jump interval
106 int sxbeg=35; //x-begining index of source, starting from 0
107 int szbeg=1; //z-begining index of source, starting from 0
108 int gxbeg=0; //x-begining index of geophone, starting from 0
109 int gzbeg=2; //z-begining index of geophone, starting from 0
110 */
111 /*
112 const bool csdgather=false; // common shot gather (CSD) or not
113 char *model_file="marm240x737.bin";
114 const int nz1=240;
115 const int nx1=737;
116 const float dx=12.5;
117 const float dz=12.5;
118 const float fm=20.0;
119 const float dt=0.001;
120 const int nt=3700;
121 const int ns=21;
122 const int ng=737;
123
124 int jsx=32;

```

```

125 int jsz=0;
126 int jgx=1;
127 int jgz=0;
128 int sxbeg=45;//x-begin point of source, index starting from 0
129 int szbeg=1;//z-begin point of source, index starting from 0
130 int gxbeg=0;//x-begin point of geophone, index starting from 0
131 int gzbeg=2;//z-begin point of geophone, index starting from 0
132
133 */
134 /*
135 char *model_file="syn320x320.bin";
136 const int      nz1=320;
137 const int      nx1=320;
138 const float    dx=5.0;
139 const float    dz=5.0;
140 const float    fm=25.0;
141 const float    dt=0.001;
142 const int      nt=1800;
143 const int      ns=11;
144 const int      ng=60;
145
146 int jsx=26;      //source x-axis jump interval
147 int jsz=0;      //source z-axis jump interval
148 int jgx=1;      //geophone x-axis jump interval
149 int jgz=0;      //geophone z-axis jump interval
150 int sxbeg=30;   //x-beginning index of source, starting from 0
151 int szbeg=1;   //z-beginning index of source, starting from 0
152 int gxbeg=0;   //x-beginning index of geophone, starting from 0
153 int gzbeg=2;   //z-beginning index of geophone, starting from 0
154 */
155
156
157 char *model_file="marm751x2301.bin";
158 const int      nz1=751;
159 const int      nx1=2301;
160 const float    dx=4.0;
161 const float    dz=4.0;
162 const float    fm=20.0;
163 const float    dt=0.0003;
164 const int      nt=13000;      // 10000
165 const int      ns=101;      // 41
166 const int      ng=301;
167
168 int jsx=20;      //source x-axis jump interval
169 int jsz=0;      //source z-axis jump interval
170 int jgx=1;      //geophone x-axis jump interval
171 int jgz=0;      //geophone z-axis jump interval
172 int sxbeg=150;  //x-beginning index of source, starting from 0
173 int szbeg=2;    //z-beginning index of source, starting from 0
174 int gxbeg=0;    //x-beginning index of geophone, starting from 0
175 int gzbeg=3;    //z-beginning index of geophone, starting from 0
176
177
178 const float    vmute=1500;
179 const float    _dx=1.0/dx;
180 const float    _dz=1.0/dz;
181 const int      nt_h=0.65*nt; // 65% points allocated on host using zero-copy
    pinned memory, the rest on device
182
183 static int      nz, nx, nnz, nnx, N;
184 static dim3     dimbbell, dimg0, dimb0;

```

```

185 static dim3      dimglr1, dimblr1, dimglr2, dimblr2; //lr=left and right
186 static dim3      dimgtb1, dimbtb1, dimgtb2, dimbtb2; //tb=top and bottom
187
188
189 // variables on host
190 float *seis, *v0, *vel, *p;
191 // variables on device
192 int *d_Sxz, *d_Gxz; // set source and geophone
    position
193 float *d_bell,*d_wlt, *d_dobs, *d_vel; // bell, wavelet, seismograms,
    velocity (vel)
194 float *d_ustp,*d_sp0, *d_sp1, *d_svx, *d_svx; // p, vx, vz for sources
195 float *d_ugp,*d_gp0, *d_gp1, *d_gvx, *d_gvx; // p, vx, vz for geophones
196 float *d_bx1, *d_bx2, *d_bz1, *d_bz2; // PML ABC coefficients for p
    and v (vx, vz)
197 float *d_convpvx, *d_convpz, *d_convvx, *d_convvz; // auxiliary variables to
    decay p and v in PML zone
198 float *d_Iss, *d_Isg, *d_I1,*d_I2; // I1: image without
    normalization; I2: normalized image;
199 float *h_boundary, *d_boundary; // boundary on host and device
200 float *ptr=NULL;
201
202 void matrix_transpose(float *matrix, int nx, int nz)
203 {
204     float *tmp=(float*)malloc(nx*nz*sizeof(float));
205     if (tmp==NULL) {printf("out_of_memory!"); exit(1);}
206     for(int iz=0; iz<nz; iz++){
207         for(int ix=0; ix<nx; ix++){
208             tmp[iz+nz*ix]=matrix[ix+nx*iz];
209         }
210     }
211     memcpy(matrix, tmp, nx*nz*sizeof(float));
212     free(tmp);
213 }
214
215
216 // a: size=nz1*nx1; b: size=nnz*nnx;
217 void expand(float *a, float *b, int npml, int nnz, int nnx, int nz1, int nx1)
218 { /*< expand domain of 'a' to 'b' >*/
219     int iz, ix;
220     for (ix=0; ix<nx1; ix++) {
221         for (iz=0; iz<nz1; iz++) {
222             b[(npml+ix)*nnz+(npml+iz)] = a[ix*nz1+iz];
223         }
224     }
225     for (ix=0; ix<nnx; ix++) {
226         for (iz=0; iz<npml; iz++) b[ix*nnz+iz] = b[ix*nnz+npml]; //top
227         for (iz=npml; iz<nnz; iz++) b[ix*nnz+iz] = b[ix*nnz+npml+nz1-1]; //
            bottom
228     }
229
230     for (iz=0; iz<nnz; iz++){
231         for(ix=0; ix<npml; ix++) b[ix*nnz+iz] = b[npml*nnz+iz]; // left
232         for (ix=npml+nx1; ix<nnx; ix++) b[ix*nnz+iz] = b[(npml+nx1-1)*nnz+iz]; //
            right
233     }
234 }
235
236 // from: size=nnz*nnx; to: size=nz1*nx1;
237 void window(float *from, float *to, int npml, int nnz, int nnx, int nz1, int nx1
    )

```

```

238 {
239     int ix, iz;
240     for(ix=0; ix<nx1; ix++){
241         for(iz=0; iz<nz1; iz++){
242             to[iz+ix*nz1]=from[(iz+npml)+(ix+npml)*nnz];
243         }
244     }
245 }
246
247
248 void check_gird_sanity()
249 {
250     float C;
251     if(NJ==2) C=1;
252     else if (NJ==4) C=0.857;
253     else if (NJ==6) C=0.8;
254     else if (NJ==8) C=0.777;
255     else if (NJ==10) C=0.759;
256
257     float maxvel=vel[0], minvel=vel[0];
258     for(int i=0; i<N; i++) {
259         if(vel[i]>maxvel) maxvel=vel[i];
260         if(vel[i]<minvel) minvel=vel[i];
261     }
262     float tmp=dt*maxvel*sqrt(1.0/(dx*dx)+1.0/(dz*dz));
263
264     if (tmp>=C) printf("Stability_condition_not_satisfied!\n");
265     if (fm>=minvel/(5*max(dx,dz))) printf("Dispersion_relation_not_satisfied!\n");
266 }
267
268
269 void device_alloc()
270 {
271     cudaMalloc(&d_bell, (2*nbell+1)*(2*nbell+1)*sizeof(float));
272     cudaMalloc(&d_Sxz, ns*sizeof(int));
273     cudaMalloc(&d_Gxz, ng*sizeof(int));
274     cudaMalloc(&d_wlt, nt*sizeof(float));
275     cudaMalloc(&d_dobs, ng*nt*sizeof(float));
276     cudaMalloc(&d_vel, N*sizeof(float));
277     cudaMalloc(&d_usp, N*sizeof(float));
278     cudaMalloc(&d_sp0, N*sizeof(float));
279     cudaMalloc(&d_sp1, N*sizeof(float));
280     cudaMalloc(&d_svx, N*sizeof(float));
281     cudaMalloc(&d_svy, N*sizeof(float));
282     cudaMalloc(&d_ugp, N*sizeof(float));
283     cudaMalloc(&d_gp0, N*sizeof(float));
284     cudaMalloc(&d_gp1, N*sizeof(float));
285     cudaMalloc(&d_gvx, N*sizeof(float));
286     cudaMalloc(&d_gvy, N*sizeof(float));
287     cudaMalloc(&d_bx1, 2*npml*nnz*sizeof(float)); // left and right ABC
288     cudaMalloc(&d_bz1, 2*npml*nnx*sizeof(float)); // top and bottom ABC
289     cudaMalloc(&d_bx2, 2*npml*nnz*sizeof(float)); // left and right ABC
290     cudaMalloc(&d_bz2, 2*npml*nnx*sizeof(float)); // top and bottom ABC
291     cudaMalloc(&d_convpx, 2*npml*nnz*sizeof(float)); // (left and right)
292     cudaMalloc(&d_convpz, 2*npml*nnx*sizeof(float)); // (top and bottom)
293     cudaMalloc(&d_convvx, 2*npml*nnz*sizeof(float)); // (left and right)

```



```

294     cudaMalloc(&d_convvz,    2*npml*nnx*sizeof(float)); // (top and bottom)
295     cudaMalloc(&d_Iss,      N*sizeof(float));
296     cudaMalloc(&d_Isg,      N*sizeof(float));
297     cudaMalloc(&d_I1,       N*sizeof(float));
298     cudaMalloc(&d_I2,       N*sizeof(float));
299     cudaHostAlloc(&h_boundary, nt_h*2*(NJ-1)*(nnx+nnz)*sizeof(float),
        cudaHostAllocMapped);
300     cudaMalloc(&d_boundary, (nt-nt_h)*2*(NJ-1)*(nnx+nnz)*sizeof(float));
301
302     cudaError_t err = cudaGetLastError ();
303     if (cudaSuccess != err)
304         printf("Cuda_error: _Failed_to_allocate_required_memory!: _%s",
            cudaGetErrorString(err));
305 }
306
307
308 void device_free ()
309 {
310     cudaFree(d_bell);
311     cudaFree(d_Sxz);
312     cudaFree(d_Gxz);
313     cudaFree(d_wlt);
314     cudaFree(d_dobs);
315     cudaFree(d_vel);
316     cudaFree(d_ustp);
317     cudaFree(d_sp0);
318     cudaFree(d_sp1);
319     cudaFree(d_svx);
320     cudaFree(d_svz);
321     cudaFree(d_ugp);
322     cudaFree(d_gp0);
323     cudaFree(d_gp1);
324     cudaFree(d_gvx);
325     cudaFree(d_gvz);
326     cudaFree(d_bx1);
327     cudaFree(d_bx2);
328     cudaFree(d_bz1);
329     cudaFree(d_bz2);
330     cudaFree(d_convpx);
331     cudaFree(d_convpz);
332     cudaFree(d_convvx);
333     cudaFree(d_convvz);
334     cudaFree(d_Iss);
335     cudaFree(d_Isg);
336     cudaFree(d_I1);
337     cudaFree(d_I2);
338     cudaFreeHost(h_boundary);
339     cudaFree(d_boundary);
340
341     cudaError_t err = cudaGetLastError ();
342     if (cudaSuccess != err)
343         printf("Cuda_error: _Failed_to_free_the_allocated_memory!: _%s",
            cudaGetErrorString(err));
344 }
345
346 //up:laplace(p)
347 void wavefield_ini(float *d_up, float *d_p0, float *d_p1, float *d_vx, float *
        d_vz, float *d_convpx, float *d_convpz, float *d_convvx, float *d_convvz)
348 {
349     cudaMemset(d_up,    0,    N*sizeof(float));
350     cudaMemset(d_p0,    0,    N*sizeof(float));

```

```

351     cudaMemset(d_p1,          0,      N*sizeof(float));
352     cudaMemset(d_vx,          0,      N*sizeof(float));
353     cudaMemset(d_vz,          0,      N*sizeof(float));
354     cudaMemset(d_convpx,      0,      2*npml*nnz*sizeof(float));
355     cudaMemset(d_convpz,      0,      2*npml*nnx*sizeof(float));
356     cudaMemset(d_convvx,      0,      2*npml*nnz*sizeof(float));
357     cudaMemset(d_convvz,      0,      2*npml*nnx*sizeof(float));
358
359     cudaError_t err = cudaGetLastError ();
360     if (cudaSuccess != err)
361         printf("Cuda_error: Failed to initialize the wavefield variables!: %s",
362             cudaGetErrorString(err));
363 }
364
365 void forward_laplacian(float *d_up, float *d_p1, float *d_vx, float *d_vz, float
366     *d_convvx, float *d_convvz, float *d_convpx, float *d_convpz, float *d_bx1,
367     float *d_bz1, float *d_bx2, float *d_bz2)
368 {
369     // p0: p{it-1};
370     // p1: p{it}; up: laplacian of p1
371     // p{it+1}-->p0
372     if (NJ==2) {
373         cuda_forward_v_2<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
374             npml, nnz, nnx);
375         cuda_PML_vz_2<<<<dimgtb1, dimbtb1>>>>(d_p1, d_convpz, d_bz2, d_vz,
376             _dz, npml, nnz, nnx);
377         cuda_PML_vx_2<<<<dimglr1, dimblr1>>>>(d_p1, d_convpx, d_bx2, d_vx,
378             _dx, npml, nnz, nnx);
379         cuda_forward_up_2<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
380             npml, nnz, nnx);
381         cuda_PML_upz_2<<<<dimgtb1, dimbtb1>>>>(d_up, d_convvz, d_bz1, d_vz,
382             _dz, npml, nnz, nnx);
383         cuda_PML_upx_2<<<<dimglr1, dimblr1>>>>(d_up, d_convvx, d_bx1, d_vx,
384             _dx, npml, nnz, nnx);
385     } else if (NJ==4){
386         cuda_forward_v_4<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
387             npml, nnz, nnx);
388         cuda_PML_vz_4<<<<dimgtb1, dimbtb1>>>>(d_p1, d_convpz, d_bz2, d_vz,
389             _dz, npml, nnz, nnx);
390         cuda_PML_vx_4<<<<dimglr1, dimblr1>>>>(d_p1, d_convpx, d_bx2, d_vx,
391             _dx, npml, nnz, nnx);
392         cuda_forward_up_4<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
393             npml, nnz, nnx);
394         cuda_PML_upz_4<<<<dimgtb1, dimbtb1>>>>(d_up, d_convvz, d_bz1, d_vz,
395             _dz, npml, nnz, nnx);
396         cuda_PML_upx_4<<<<dimglr1, dimblr1>>>>(d_up, d_convvx, d_bx1, d_vx,
397             _dx, npml, nnz, nnx);
398     } else if (NJ==6){
399         cuda_forward_v_6<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
400             npml, nnz, nnx);
401         cuda_PML_vz_6<<<<dimgtb1, dimbtb1>>>>(d_p1, d_convpz, d_bz2, d_vz,
402             _dz, npml, nnz, nnx);
403         cuda_PML_vx_6<<<<dimglr1, dimblr1>>>>(d_p1, d_convpx, d_bx2, d_vx,
404             _dx, npml, nnz, nnx);
405         cuda_forward_up_6<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
406             npml, nnz, nnx);
407         cuda_PML_upz_6<<<<dimgtb1, dimbtb1>>>>(d_up, d_convvz, d_bz1, d_vz,
408             _dz, npml, nnz, nnx);
409         cuda_PML_upx_6<<<<dimglr1, dimblr1>>>>(d_up, d_convvx, d_bx1, d_vx,
410             _dx, npml, nnz, nnx);
411     } else if (NJ==8){

```

```

391     cuda_forward_v_8<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
392         npml, nnz, nnx);
393     cuda_PML_vz_8<<<<dimgtb1, dimb1>>>>(d_p1, d_convvpz, d_bz2, d_vz,
394         _dz, npml, nnz, nnx);
395     cuda_PML_vx_8<<<<dimglr1, dimblr1>>>>(d_p1, d_convvpz, d_bx2, d_vx,
396         _dx, npml, nnz, nnx);
397     cuda_forward_up_8<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
398         npml, nnz, nnx);
399     cuda_PML_upz_8<<<<dimgtb1, dimb1>>>>(d_up, d_convvpz, d_bz1, d_vz,
400         _dz, npml, nnz, nnx);
401     cuda_PML_upx_8<<<<dimglr1, dimblr1>>>>(d_up, d_convvpz, d_bx1, d_vx,
402         _dx, npml, nnz, nnx);
403 } else if (NJ==10){
404     cuda_forward_v_10<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
405         npml, nnz, nnx);
406     cuda_PML_vz_10<<<<dimgtb1, dimb1>>>>(d_p1, d_convvpz, d_bz2, d_vz,
407         _dz, npml, nnz, nnx);
408     cuda_PML_vx_10<<<<dimglr1, dimblr1>>>>(d_p1, d_convvpz, d_bx2, d_vx,
409         _dx, npml, nnz, nnx);
410     cuda_forward_up_10<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
411         npml, nnz, nnx);
412     cuda_PML_upz_10<<<<dimgtb1, dimb1>>>>(d_up, d_convvpz, d_bz1,
413         d_vz, _dz, npml, nnz, nnx);
414     cuda_PML_upx_10<<<<dimglr1, dimblr1>>>>(d_up, d_convvpz, d_bx1,
415         d_vx, _dx, npml, nnz, nnx);
416 }
417 }
418
419 void backward_laplacian(float *d_up, float *d_p1, float *d_vx, float *d_vz)
420 {
421     // p0: p{it-1};
422     // p1: p{it}; up: laplacian of p1
423     // p{it+1}-->p0
424     if (NJ==2){
425         cuda_forward_v_2<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
426             npml, nnz, nnx);
427         cuda_forward_up_2<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
428             npml, nnz, nnx);
429     }
430     else if (NJ==4) {
431         cuda_forward_v_4<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
432             npml, nnz, nnx);
433         cuda_forward_up_4<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
434             npml, nnz, nnx);
435     }
436     else if (NJ==6) {
437         cuda_forward_v_6<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
438             npml, nnz, nnx);
439         cuda_forward_up_6<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
440             npml, nnz, nnx);
441     }
442     else if (NJ==8) {
443         cuda_forward_v_8<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
444             npml, nnz, nnx);
445         cuda_forward_up_8<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
446             npml, nnz, nnx);
447     }
448     else if (NJ==10){
449         cuda_forward_v_10<<<<dimg0, dimb0>>>>(d_p1, d_vx, d_vz, _dx, _dz,
450             npml, nnz, nnx);

```

```

430         cuda_forward_up_10<<<<dimg0, dimb0>>>>(d_up, d_vx, d_vz, _dx, _dz,
431             npml, nnz, nnx);
432     }
433 }
434
435 int main(int argc, char *argv[])
436 {
437     nx=(int)((nx1+Block_Size1-1)/Block_Size1)*Block_Size1;
438     nz=(int)((nz1+Block_Size2-1)/Block_Size2)*Block_Size2;
439     nnz = 2*npml+nz;
440     nnx = 2*npml+nx;
441     N=nnz*nnx;
442     dimbbell=dim3(2*nbell+1,2*nbell+1);
443     dimb0=dim3(Block_Size1, Block_Size2);    dimg0=dim3(nnz/Block_Size1, nnx/
         Block_Size2);
444     dimblr1=dim3(Block_Size1, 32);            dimglr1=dim3(nnz/Block_Size1, 2)
         ;
445     dimbtb1=dim3(32, Block_Size2);            dimgtb1=dim3(2, nnx/Block_Size2)
         ;
446     dimblr2=dim3(nnz/Block_Size1, (NJ+15)/16); dimglr2=dim3(Block_Size1, 16);
447     dimbtb2=dim3(16, Block_Size2);            dimgtb2=dim3((NJ+15)/16, nnx/
         Block_Size2);
448
449     v0=(float*)malloc(nx1*nz1*sizeof(float));
450     if (v0==NULL) { printf("out_of_memory!"); exit(1);}
451     seis=(float*)malloc(ng*nt*sizeof(float));
452     if (seis==NULL) { printf("out_of_memory!"); exit(1);}
453     vel=(float*)malloc(N*sizeof(float));
454     if (vel==NULL) { printf("out_of_memory!"); exit(1);}
455     p=(float*)malloc(N*sizeof(float));
456     if (p==NULL) { printf("out_of_memory!"); exit(1);}
457     memset(v0, 0, nz1*nx1*sizeof(float));
458     memset(seis, 0, ng*nt*sizeof(float));
459     memset(vel, 0, N*sizeof(float));
460     memset(p, 0, N*sizeof(float));
461
462     FILE *fp,*fp1;
463     fp=fopen(model_file,"rb");
464     if (fp==NULL) { printf("cannot_open_the_model_file"); exit(1);}
465     fread(v0, sizeof(float), nz1*nx1, fp);
466     fclose(fp);
467     expand(v0, vel, npml, nnz, nnx, nz1, nx1);
468
469
470     printf("NJ=%d\t\n", NJ);
471     printf("npml=%d\t\n", npml);
472     printf("nx1=%d\t\n", nx1);
473     printf("nz1=%d\t\n", nz1);
474     printf("nx=%d\t\n", nx);
475     printf("nz=%d\t\n", nz);
476     printf("nnx=%d\t\n", nnx);
477     printf("nnz=%d\t\n", nnz);
478     printf("dx=%g\t\t(m)\n", dx);
479     printf("dz=%g\t\t(m)\n", dz);
480     printf("dt=%g\t\t(s)\n", dt);
481     printf("fm=%g\t\t(Hz)\n", fm);
482     printf("nt=%d\n", nt);
483     printf("ns=%d\n", ns);
484     printf("ng=%d\n", ng);
485     check_gird_sanity();

```

```

486     cudaSetDevice(0);
487     cudaError_t err = cudaGetLastError ();
488     if (cudaSuccess != err)
489         printf("Cuda_error:_Failed_to_initialize_device:_%s", cudaGetErrorString
490             (err));
491     device_alloc();
492
493     float mstimer = 0; // timer unit: millionseconds
494     cudaEvent_t start, stop;
495     cudaEventCreate(&start);
496     cudaEventCreate(&stop);
497
498     fp=fopen("img1.bin","wb");
499     if (fp==NULL) { printf("cannot_open_the_file\n"); exit(1);}
500     fp1=fopen("img2.bin","wb");
501     if (fp1==NULL) { printf("cannot_open_the_file\n"); exit(1);}
502
503     cuda_ini_bell<<<<dim3(1,1),dim3(2*nbell+1,2*nbell+1)>>>(d_bell);
504     cuda_ricker_wavelet<<<<(nt+511)/512,512>>>(d_wlt, fm, dt, nt);
505     if (!(sxbeg>=0 && szbeg>=0 && sxbeg+(ns-1)*jsx<nx && szbeg+(ns-1)*jsz<nz
506         ))
507     { printf("sources_exceeds_the_computing_zone!\n"); exit(1);}
508     cuda_set_sg<<<<(ns+255)/256, 256>>>(d_Sxz, sxbeg, szbeg, jsx, jsz, ns,
509         npml, nnz);
510
511     int distx=sxbeg-gxbeg;
512     int distz=szbeg-gzbeg;
513     if (csdgather) {
514         //distance between source and geophone at the beginning
515         if (!(gxbeg>=0 && gzbeg>=0 && gxbeg+(ng-1)*jgx<nx && gzbeg+(ng
516             -1)*jgz<nz &&
517             (sxbeg+(ns-1)*jsx)+(ng-1)*jgx-distx <nx && (szbeg+(ns-1)*jsz)+(
518                 ng-1)*jgz-distz <nz))
519             { printf("geophones_exceeds_the_computing_zone!\n"); exit(1);}
520     }
521     else{
522         if (!(gxbeg>=0 && gzbeg>=0 && gxbeg+(ng-1)*jgx<nx && gzbeg+(ng
523             -1)*jgz<nz))
524             { printf("geophones_exceeds_the_computing_zone!\n"); exit(1);}
525     }
526     cuda_set_sg<<<<(ng+255)/256, 256>>>(d_Gxz, gxbeg, gzbeg, jgx, jgz, ng,
527         npml, nnz);
528
529     //cuda_set2<<<<dimg0, dimb0>>>(d_vel, 1700.0, 2000.0,nnz, nnx);
530     cudaMemcpy(d_vel, vel, N*sizeof(float), cudaMemcpyHostToDevice);
531     cudaMemset(d_Iss, 0, N*sizeof(float));
532     cudaMemset(d_Isg, 0, N*sizeof(float));
533     cudaMemset(d_I1, 0, N*sizeof(float));
534     cudaMemset(d_I2, 0, N*sizeof(float));
535     cuda_ini_abcz<<<<dimgtb1, dimbtb1>>>(d_vel, d_bz1, d_bz2, dx, dz, dt,
536         npml, nnz, nnx);
537     cuda_ini_abcx<<<<dimglr1, dimblr1>>>(d_vel, d_bx1, d_bx2, dx, dz, dt,
538         npml, nnz, nnx);
539
540     for(int is=0; is<ns; is++)
541     {
542         cudaEventRecord(start);
543
544         cudaMemset(d_Isg, 0, N*sizeof(float));
545         cudaMemset(d_Iss, 0, N*sizeof(float));

```

```

538     cudaMemset(d_dobs, 0, nt*ng*sizeof(float));
539     cudaMemset(h_boundary, 0, nt*h*2*(NJ-1)*(nnx+nnz)*sizeof(
        float));
540     cudaMemset(d_boundary, 0, (nt-nt.h)*2*(NJ-1)*(nnx+nnz)*
        sizeof(float));
541     wavefield_ini(d_ush, d_sp0, d_sp1, d_svx, d_svz, d_convpx,
        d_convpz, d_convvx, d_convvz);
542     if (csdgather) {
543         gxbeg=sxbeg+is*jsx-distx;
544         cuda_set_sg<<<(ng+255)/256, 256>>>(d_Gxz, gxbeg, gzbeg,
            jgx, jgz, ng, npml, nnz);
545     }
546     for(int kt=0; kt<nt; kt++)
547     {
548         forward_laplacian(d_ush, d_sp1, d_svx, d_svz, d_convvx,
            d_convvz, d_convpx, d_convpz, d_bx1, d_bz1, d_bx2,
            d_bz2);
549         cuda_step_forward<<<dimg0,dimb0>>>(d_vel, d_ush, d_sp0,
            d_sp1, dt, false, npml, nnz, nnx);
550         cuda_add_bellwlt<<<dim3(1,1), dimbbell>>>(d_sp1, d_bell,
            &d_wlt[kt], &d_Sxz[is], 1, npml, nnz, nnx, true);
551         ptr=d_sp0; d_sp0=d_sp1; d_sp1=ptr;
552
553         cuda_record<<<(ng+255)/256, 256>>>(d_sp0, &d_dobs[kt*ng
            ], d_Gxz, ng);
555     /*
556         if (kt%50==0){
557             cudaMemcpy(p, d_sp1, N*sizeof(float),
                cudaMemcpyDeviceToHost);
558             fwrite(p, sizeof(float), N, fp);
559         }
560     */
561         cuda_mute<<<(ng+511)/512, 512>>>(&d_dobs[kt*ng], gzbeg,
            szbeg, gxbeg, sxbeg+is*jsx, jgx, kt, 280, vmute, dt,
            dz, dx, ng);
562
563         if(kt<nt.h) cudaHostGetDevicePointer(&ptr, &h_boundary[
            kt*2*(NJ-1)*(nnx+nnz)], 0);
564         else ptr=&d_boundary[(kt-nt.h)*2*(NJ-1)*(nnx+nnz)];
565         cuda_rw_boundarytb<<<dimgtb2, dimbtb2>>>(ptr, d_sp0,
            npml, nnz, nnx, NJ, false);
566         cuda_rw_boundarylr<<<dimglr2, dimblr2>>>(&ptr[2*(NJ-1)*
            nnx], d_sp0, npml, nnz, nnx, NJ, false);
567     }
568     /*
569     cudaMemcpy(seis, d_dobs, nt*ng*sizeof(float),
        cudaMemcpyDeviceToHost);
570     matrix_transpose(seis, ng, nt); // before: nx=ng; nz=nt; after:
        nx=nt; nz=ng;
571     fwrite(seis, sizeof(float), nt*ng, fp);
572     */
573
574     ptr=d_sp0; d_sp0=d_sp1; d_sp1=ptr;
575     wavefield_ini(d_ugp, d_gp0, d_gp1, d_gvx, d_gvz, d_convpx,
        d_convpz, d_convvx, d_convvz);
576     for(int kt=nt-1; kt>-1; kt--)
577     {
578         // read saved boundary
579         if(kt<nt.h) cudaHostGetDevicePointer(&ptr, &h_boundary[
            kt*2*(NJ-1)*(nnx+nnz)], 0);

```

```

580     else ptr=&d_boundary[(kt-nt.h)*2*(NJ-1)*(nnx+nnz)];
581     cuda_rwlock_boundarytb<<<<dimgtb2, dimbtb2>>>>(ptr,
582         d_sp1, npml, nnz, nnx, NJ, true);
583     cuda_rwlock_boundarylr<<<<dimglr2, dimblr2>>>>(&ptr[2*(NJ-1)*
584         nnx], d_sp1, npml, nnz, nnx, NJ, true);
585     // subtract the wavelet
586     cuda_add_bellwlt<<<<dim3(1,1), dimbbell>>>>(d_sp1, d_bell,
587         &d_wlt[kt], &d_Sxz[is], 1, npml, nnz, nnx, false);
588     backward_laplacian(d_ush, d_sp1, d_svx, d_svy);
589     // backward time step source wavefield
590     cuda_step_forward<<<<dimg0,dimb0>>>>(d_vel, d_ush, d_sp0,
591         d_sp1, dt, false, npml, nnz, nnx);
592     ptr=d_sp0; d_sp0=d_sp1; d_sp1=ptr;
593
594     // backward time step receiver wavefield
595     forward_laplacian(d_ugp, d_gp1, d_gvx, d_gvy, d_convvx,
596         d_convvy, d_convpx, d_convpz, d_bx1, d_bz1, d_bx2,
597         d_bz2);
598     cuda_step_forward<<<<dimg0,dimb0>>>>(d_vel, d_ugp, d_gp0,
599         d_gp1, dt, false, npml, nnz, nnx);
600     // add receiver term
601     cuda_add_source<<<<(ng+255)/256,256>>>>(d_gp0, &d_dobs[kt*
602         ng], d_Gxz, ng, true);
603     ptr=d_gp0; d_gp0=d_gp1; d_gp1=ptr;
604
605     cuda_cross_correlate<<<<dimg0, dimb0>>>>(d_Isg, d_Iss,
606         d_sp0, d_gp0, npml, nnz, nnx);
607 }
608 cuda_imaging<<<<dimg0, dimb0>>>>(d_Isg, d_Iss, d_I1, d_I2, npml,
609     nnz, nnx);
610
611     cudaEventRecord(stop);
612     cudaEventSynchronize(stop);
613     cudaEventElapsedTime(&mstimer, start, stop);
614     printf("%d_shot_finished: %f(s)\n", is+1, mstimer*1e-3);
615 }
616 cuda_laplace_filter<<<<dimg0,dimb0>>>>(d_I1, d_ush, _dz, _dx, npml, nnz, nnx)
617 ;
618 cuda_laplace_filter<<<<dimg0,dimb0>>>>(d_I2, d_ugp, _dz, _dx, npml, nnz, nnx)
619 ;
620 cudaMemcpy(p, d_ush, N*sizeof(float), cudaMemcpyDeviceToHost);
621 window(p, v0, npml, nnz, nnx, nz1, nx1);
622 fwrite(v0, sizeof(float), nz1*nx1, fp);
623 fclose(fp);
624 cudaMemcpy(p, d_ugp, N*sizeof(float), cudaMemcpyDeviceToHost);
625 window(p, v0, npml, nnz, nnx, nz1, nx1);
626 fwrite(v0, sizeof(float), nz1*nx1, fp1);
627 fclose(fp1);
628
629     cudaEventDestroy(start);
630     cudaEventDestroy(stop);
631
632     free(seis);
633     free(v0);
634     free(vel);
635     free(p);
636     device_free();
637
638     return 0;
639 }

```



```

1  /* cuda_kernels.cu
2  Copyright (C) 2013 Xi'an Jiaotong University (Pengliang Yang)
3  Email: ypl.2100@gmail.com
4  Acknowledgement: This code is written with the help of Baoli Wang.
5  */
6  __global__ void cuda_set(float*p, float c, int nnz, int nnx)
7  {
8      int i1=threadIdx.x+blockDim.x*blockIdx.x;
9      int i2=threadIdx.y+blockDim.y*blockIdx.y;
10     int id=i1+i2*nnz;
11
12     if (i1>=0 && i1<nnz && i2>=0 && i2<nnx) p[id]=c;
13 }
14
15 // set part of p[nnz][nnx]=c1, part of p[nnz][nnx]=c2;
16 __global__ void cuda_set2(float*p, float c1, float c2, int nnz, int nnx)
17 {
18     int i1=threadIdx.x+blockDim.x*blockIdx.x;
19     int i2=threadIdx.y+blockDim.y*blockIdx.y;
20     int id=i1+i2*nnz;
21
22     if (i1<nnz/2 && i2<nnx) p[id]=c1;
23     else p[id]=c2;
24 }
25
26 // set the positions of sources and geophones in whole domain
27 __global__ void cuda_set_sg(int *sxz, int sxbeg, int szbeg, int jsx, int jsz,
28     int ns, int npml, int nnz)
29 {
30     int id=threadIdx.x+blockDim.x*blockIdx.x;
31     if (id<ns) sxz[id]=nnz*(sxbeg+id*jsx+npml)+(szbeg+id*jsz+npml);
32 }
33
34 // generate ricker wavelet with time deley
35 __global__ void cuda_ricker_wavelet(float *wlt, float fm, float dt, int nt)
36 {
37     int it=threadIdx.x+blockDim.x*blockIdx.x;
38     float tmp = PI*fm*fabsf(it*dt-1.0/fm); //delay the wavelet to exhibit
39     // all waveform
40     tmp *=tmp;
41     if (it<nt) wlt[it]= (1.0-2.0*tmp)*expf(-tmp); // ricker wavelet at
42     // time: t=nt*dt
43 }
44
45 // add==true, add (inject) the source; add==false, subtract the source
46 __global__ void cuda_add_source(float *p, float *source, int *Sxz, int ns, bool
47     add)
48 {
49     int id=threadIdx.x+blockDim.x*blockIdx.x;
50     if (id<ns)
51     {
52         if (add) p[Sxz[id]]+=source[id];
53         else p[Sxz[id]]-=source[id];
54     }
55 }
56
57 // record the seismogram at time kt
58 __global__ void cuda_record(float*p, float *seis_kt, int *Gxz, int ng)
59 {

```



```

57         int id=threadIdx.x+blockDim.x*blockIdx.x;
58         if (id<ng) seis_kt[id]=p[Gxz[id]];
59     }
60
61     // mute the direct arrival according to the given velocity vmute
62     __global__ void cuda_mute(float *seis_kt, int gzbeg, int szbeg, int gxbeg, int
        sxc, int jgx, int kt, int ntd, float vmute, float dt, float dz, float dx, int
        ng)
63     {
64         int id=threadIdx.x+blockDim.x*blockIdx.x;
65         float a=dx*abs(gxbeg+id*jgx-sxc);
66         float b=dz*(gzbeg-szbeg);
67         float t0=sqrtf(a*a+b*b)/vmute;
68         int ktt=int(t0/dt)+ntd; // ntd is manually added to obtain the best
            muting effect.
69         if (id<ng && kt<ktt) seis_kt[id]=0.0;
70     }
71
72     // initialize the PML coefficients along x direction
73     __global__ void cuda_ini_abcx(float *vel, float *bx1, float *bx2, float dx,
        float dz, float dt, int npml, int nnz, int nnx)
74     {
75         // bx1: left and right PML ABC coefficients, decay p (px,pz) along x
            direction
76         // bx2: left and right PML ABC coefficients, decay v (vx,vz) along x
            direction
77         // only 2 blocks used horizontally, blockIdx.x=0, 1
78
79         // id: position in top or bottom PML zone itself
80         // blockIdx.x==0, left PML zone; blockIdx.x==1, right PML zone
81         int i1=threadIdx.x+blockDim.x*blockIdx.x;
82         int i2=blockIdx.y*npml+threadIdx.y;
83         int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
84         int ik=i1+nnz*i2;
85
86         float Rc=1.0e-5f;
87         float d=npml*MAX(dx,dz);
88         float d0=-3.0f*vel[id]*logf(Rc)/d/2.0f;
89         float tmp1, tmp2;
90
91         if (i2<npml) // left PML zone
92         {
93             tmp1=(float)(npml-i2);
94             tmp2=tmp1-0.5f;
95
96         }
97         else // right PML zone
98         {
99             tmp1=i2-npml+0.5f;
100            tmp2=(tmp1+0.5f);
101        }
102        tmp1=tmp1/npml;
103        tmp2=tmp2/npml;
104        tmp1=tmp1*tmp1;
105        tmp2=tmp2*tmp2;
106        bx1[ik]=expf(-d0*tmp1*dt);
107        bx2[ik]=expf(-d0*tmp2*dt);
108    }
109
110
111    // initialize the PML coefficients along z-axis

```

```

112 --global__ void cuda_ini_abcz(float *vel, float *bz1, float *bz2, float dx,
113     float dz, float dt, int npml, int nnz, int nnx)
114 {
115     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
116     // direction
117     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
118     // direction
119     // only 2 blocks used vertically, blockIdx.y=0, 1
120
121     // id: position in top or bottom PML zone itself
122     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
123
124     int i1=threadIdx.x+blockIdx.x*npml;
125     int i2=threadIdx.y+blockIdx.y*blockDim.y;
126     int id=nnz*i2+(blockIdx.x*(nnz-npml)+threadIdx.x);
127     int ik=i1+2*npml*i2;
128
129     float Rc=1.0e-5f;
130     float d=npml*MAX(dx,dz);
131     float d0=-3.0f*vel[id]*logf(Rc)/d/2.0f;
132     float tmp1, tmp2;
133
134     if (i1<npml) // top PML zone
135     {
136         tmp1=(float)(npml-i1);
137         tmp2=tmp1-0.5f;
138     }
139     else // bottom PML zone
140     {
141         tmp1=i1-npml+0.5f;
142         tmp2=(float)(tmp1+0.5f);
143     }
144     tmp1=tmp1/npml;
145     tmp2=tmp2/npml;
146     tmp1=tmp1*tmp1;
147     tmp2=tmp2*tmp2;
148     bz1[ik]=expf(-d0*tmp1*dt);
149     bz2[ik]=expf(-d0*tmp2*dt);
150 }
151
152 --global__ void cuda_ini_bell(float *bell)
153 {
154     int i1=threadIdx.x;
155     int i2=threadIdx.y;
156     int id=i1+i2*(2*nbell+1);
157     float s = 0.5*nbell;
158     bell[id]=expf(-((i1-nbell)*(i1-nbell)+(i2-nbell)*(i2-nbell))/s);
159 }
160
161 // inject Gaussian bell smoothed wavelet
162 // lauch configuration: <<<<dim3(ns,1), dim3(2*nbell+1,2*nbell+1)>>>>
163 // add==true, add (inject) the wavelet; add==false, subtract the wavelet
164 --global__ void cuda_add_bellwlt(float *p, float *bell, float *wlt, int *Sxz,
165     int ns, int npml, int nnz, int nnx, bool add)
166 {
167     int i1=threadIdx.x;
168     int i2=threadIdx.y;
169     int is=blockIdx.x; // source wavelet index
170
171     if (is<ns)

```

```

169     {
170         if (add)          p[Sxz[is]+(i1-nbell)+(i2-nbell)*nnz]+=bell[i1+i2
                             *(2*nbell+1)]*wlt[is];
171         else             p[Sxz[is]+(i1-nbell)+(i2-nbell)*nnz]-=bell[i1+i2
                             *(2*nbell+1)]*wlt[is];
172     }
173 }
174
175 //===== NJ=2
176 //=====
177 --global-- void cuda_forward_v_2(float *p, float *vx, float *vz, float _dx,
178     float _dz, int npml, int nnz, int nnx)
179 {
180     int i1=blockIdx.x*blockDim.x+threadIdx.x;
181     int i2=blockIdx.y*blockDim.y+threadIdx.y;
182     int id=i1+i2*nnz;
183
184     __shared__ float s_p[Block.Size1+1][Block.Size2+1];
185     s_p[threadIdx.x][threadIdx.y]=p[id];
186     if (threadIdx.x>blockDim.x-2)
187     {
188         if (blockIdx.x<gridDim.x-1)      s_p[threadIdx.x+1][threadIdx.y]=
189             p[id+1];
190         else                               s_p[threadIdx.x+1][threadIdx.y
191             ]=0.0f;
192     }
193     if (threadIdx.y>blockDim.y-2)
194     {
195         if (blockIdx.y<gridDim.y-1)      s_p[threadIdx.x][threadIdx.y+1]=
196             p[id+nnz];
197         else                               s_p[threadIdx.x][threadIdx.y
198             +1]=0.0f;
199     }
200     __syncthreads();
201
202     float diff1=(s_p[threadIdx.x+1][threadIdx.y]-s_p[threadIdx.x][threadIdx.
203         y]); // .x→1st dim→ i1
204     float diff2=(s_p[threadIdx.x][threadIdx.y+1]-s_p[threadIdx.x][threadIdx.
205         y]); // .y→2nd dim→ i2
206     vz[id]=-dz*diff1;
207     vx[id]=-dx*diff2;
208 }
209
210 --global-- void cuda_PML_vz_2(float *p, float *convpz, float *bz, float *vz,
211     float _dz, int npml, int nnz, int nnx)
212 {
213     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
214     // direction
215     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
216     // direction
217     // only 2 blocks used vertically, blockIdx.y=0,1
218
219     // id: position in whole zone(including PML)
220     // ik: position in top or bottom PML zone itself
221     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
222     int i1=threadIdx.x+blockIdx.x*npml;
223     int i2=threadIdx.y+blockIdx.y*blockDim.y;
224     int ik=i1+2*npml*i2;
225     int id=(blockIdx.x*(nnz-npml)+threadIdx.x)+nnz*i2;
226
227     __shared__ float s_p[33][Block.Size2];

```

```

217     s_p[threadIdx.x][threadIdx.y]=p[id];
218     if (threadIdx.x>30)
219     {
220         if (blockIdx.x<gridDim.x-1)         s_p[threadIdx.x+1][threadIdx.y]=
                p[id+1];
221         else                                 s_p[threadIdx.x+1][threadIdx.y
                ]=0.0f;
222     }
223     __syncthreads();
224
225     float diff1=(s_p[threadIdx.x+1][threadIdx.y]-s_p[threadIdx.x][threadIdx.
        y]);
226     convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
227     vz[id]+=convpz[ik];
228 }
229
230 __global__ void cuda_PML_vx_2(float *p, float *convpx, float *bx, float *vx,
    float _dx, int npml, int nnz, int nnx)
231 {
232     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
        direction
233     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
        direction
234     // only 2 blocks used vertically, blockIdx.y=0, 1
235
236     // id: position in whole zone(including PML)
237     // ik: position in top or bottom PML zone itself
238     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
239     int i1=threadIdx.x+blockIdx.x*blockDim.x;
240     int i2=threadIdx.y+blockIdx.y*npml;
241     int ik=i1+nnz*i2;
242     int id=i1+nnz*(blockIdx.y*(nnx-npml)+ threadIdx.y);
243
244     __shared__ float s_p[Block_Size1][33];
245     s_p[threadIdx.x][threadIdx.y]=p[id];
246     if (threadIdx.y>30)
247     {
248         if (blockIdx.y<gridDim.y-1)         s_p[threadIdx.x][threadIdx.y+1]=
                p[id+nnz];
249         else                                 s_p[threadIdx.x][threadIdx.y
                +1]=0.0f;
250     }
251     __syncthreads();
252
253     float diff2=(s_p[threadIdx.x][threadIdx.y+1]-s_p[threadIdx.x][threadIdx.
        y]);
254     convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
255     vx[id]+=convpx[ik];
256 }
257
258 __global__ void cuda_forward_up_2(float *up, float *vx, float *vz, float _dx,
    float _dz, int npml, int nnz, int nnx)
259 {
260     int i1=threadIdx.x+blockIdx.x*blockDim.x;
261     int i2=threadIdx.y+blockIdx.y*blockDim.y;
262     int id=i1+i2*nnz;
263
264     __shared__ float s_v1[Block_Size1+1][Block_Size2];
265     __shared__ float s_v2[Block_Size1][Block_Size2+1];
266     s_v1[threadIdx.x+1][threadIdx.y]=vz[id];
267     s_v2[threadIdx.x][threadIdx.y+1]=vx[id];

```

```

268     if (threadIdx.x<1)
269     {
270         if (blockIdx.x)                s_v1[threadIdx.x][threadIdx.y]=
            vz[id-1];
271         else                            s_v1[threadIdx.x][threadIdx.y]
            ]=0.0f;
272     }
273     if (threadIdx.y<1)
274     {
275         if (blockIdx.y)                s_v2[threadIdx.x][threadIdx.y]=
            vx[id-nnz];
276         else                            s_v2[threadIdx.x][threadIdx.y]
            ]=0.0f;
277     }
278     __syncthreads();
279
280     float diff1=(s_v1[threadIdx.x+1][threadIdx.y]-s_v1[threadIdx.x][
            threadIdx.y]);
281     float diff2=(s_v2[threadIdx.x][threadIdx.y+1]-s_v2[threadIdx.x][
            threadIdx.y]);
282     up[id]=_dz*diff1+_dx*diff2;
283 }
284
285
286
287 --global__ void cuda_PML_upz_2(float *up,   float *convvz, float *bz, float *vz,
            float _dz, int npml, int nnz, int nnx)
288 {
289     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
290     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
291     // only 2 blocks used vertically, blockIdx.y=0, 1
292
293     // id: position in whole zone(including PML)
294     // ik: position in top or bottom PML zone itself
295     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
296     int i1=threadIdx.x+blockIdx.x*npml;
297     int i2=threadIdx.y+blockIdx.y*blockDim.y;
298     int ik=i1+2*npml*i2;
299     int id=(blockIdx.x*(nnz-npml)+ threadIdx.x)+ nnz* i2;
300
301     __shared__ float s_v1[33][Block_Size2];
302     s_v1[threadIdx.x+1][threadIdx.y]=vz[id];
303     if (threadIdx.x<1)
304     {
305         if (blockIdx.x)                s_v1[threadIdx.x][threadIdx.y]=
            vz[id-1];
306         else                            s_v1[threadIdx.x][threadIdx.y]
            ]=0.0f;
307     }
308     __syncthreads();
309
310     float diff1=(s_v1[threadIdx.x+1][threadIdx.y]-s_v1[threadIdx.x][
            threadIdx.y]);
311     convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
312     up[id]+=convvz[ik];
313 }
314
315 --global__ void cuda_PML_upx_2(float *up,   float *convvx, float *bx, float *vx,
            float _dx, int npml, int nnz, int nnx)

```

```

316 {
317     // bz1: top and bottom PML ABC coefficients , decay p (px,pz) along z
           direction
318     // bz2: top and bottom PML ABC coefficients , decay v (vx,vz) along z
           direction
319     // only 2 blocks used vertically , blockIdx.y=0, 1
320
321     // id: position in whole zone(including PML)
322     // ik: position in top or bottom PML zone itself
323     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
324     int i1=threadIdx.x+blockIdx.x*blockDim.x;
325     int i2=threadIdx.y+blockIdx.y*npml;
326     int ik=i1+nnz*i2;
327     int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
328
329     __shared__ float s_v2[Block_Size1][33];
330     s_v2[threadIdx.x][threadIdx.y+1]=vx[id];
331     if (threadIdx.y<1)
332     {
333         if (blockIdx.y)                s_v2[threadIdx.x][threadIdx.y]=
           vx[id-nnz];
334         else                s_v2[threadIdx.x][threadIdx.y
           ]=0.0f;
335     }
336     __syncthreads();
337
338     float diff2=(s_v2[threadIdx.x][threadIdx.y+1]-s_v2[threadIdx.x][
           threadIdx.y]);
339     convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
340     up[id]+=convvx[ik];
341 }
342
343
344 //===== NJ=4
           =====
345 __global__ void cuda_forward_v_4(float *p, float *vx, float *vz, float _dx,
           float _dz, int npml, int nnz, int nnx)
346 {
347     int i1=blockIdx.x*blockDim.x+threadIdx.x;
348     int i2=blockIdx.y*blockDim.y+threadIdx.y;
349     int id=i1+i2*nnz;
350
351     __shared__ float s_p[Block_Size1+3][Block_Size2+3];
352     s_p[threadIdx.x+1][threadIdx.y+1]=p[id];
353     if (threadIdx.x<1)
354     {
355         if (blockIdx.x)                s_p[threadIdx.x][threadIdx.y+1]=
           p[id-1];
356         else                s_p[threadIdx.x][threadIdx.y
           +1]=0.0f;
357     }
358     if (threadIdx.x>blockDim.x-3)
359     {
360         if (blockIdx.x<gridDim.x-1)    s_p[threadIdx.x+3][threadIdx.y
           +1]=p[id+2];
361         else                s_p[threadIdx.x+3][threadIdx.y
           +1]=0.0f;
362     }
363     if (threadIdx.y<1)
364     {

```

```

365         if (blockIdx.y)                s_p[threadIdx.x+1][threadIdx.y]=
            p[id-nnz];
366         else                            s_p[threadIdx.x+1][threadIdx.y
            ]=0.0f;
367     }
368     if (threadIdx.y>blockDim.y-3)
369     {
370         if (blockIdx.y<gridDim.y-1)    s_p[threadIdx.x+1][threadIdx.y
            +3]=p[id+2*nnz];
371         else                            s_p[threadIdx.x+1][threadIdx.y
            +3]=0.0f;
372     }
373     __syncthreads();
374
375     float diff1=1.125f*(s_p[threadIdx.x+2][threadIdx.y+1]-s_p[threadIdx.x
        +1][threadIdx.y+1])
376             -0.0416666666666667f*(s_p[threadIdx.x+3][threadIdx.y+1]-
            s_p[threadIdx.x][threadIdx.y+1]);
377     float diff2=1.125f*(s_p[threadIdx.x+1][threadIdx.y+2]-s_p[threadIdx.x
        +1][threadIdx.y+1])
378             -0.0416666666666667f*(s_p[threadIdx.x+1][threadIdx.y+3]-
            s_p[threadIdx.x+1][threadIdx.y]);
379     vz[id]=_dz*diff1;
380     vx[id]=_dx*diff2;
381 }
382
383 --global-- void cuda_PML_vz_4(float *p, float *convpz, float *bz, float *vz,
        float _dz, int npml, int nnz, int nnx)
384 {
385     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
        direction
386     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
        direction
387     // only 2 blocks used vertically, blockIdx.y=0, 1
388
389     // id: position in whole zone(including PML)
390     // ik: position in top or bottom PML zone itself
391     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
392     int i1=threadIdx.x+blockIdx.x*npml;
393     int i2=threadIdx.y+blockIdx.y*blockDim.y;
394     int ik=i1+2*npml*i2;
395     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
396
397     __shared__ float s_p[35][Block_Size2];
398     s_p[threadIdx.x+1][threadIdx.y]=p[id];
399     if (threadIdx.x<1)
400     {
401         if (blockIdx.x)                s_p[threadIdx.x][threadIdx.y]=p[
            id-1];
402         else                            s_p[threadIdx.x][threadIdx.y
            ]=0.0f;
403     }
404     if (threadIdx.x>29)
405     {
406         if (blockIdx.x<gridDim.x-1)    s_p[threadIdx.x+3][threadIdx.y]=
            p[id+2];
407         else                            s_p[threadIdx.x+3][threadIdx.y
            ]=0.0f;
408     }
409     __syncthreads();
410

```

```

411         float diff1=1.125f*(s_p[threadIdx.x+2][threadIdx.y]-s_p[threadIdx.x+1][
412             threadIdx.y])
413             -0.0416666666666667f*(s_p[threadIdx.x+3][threadIdx.y]-s_p
414                 [threadIdx.x][threadIdx.y]);
415         convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
416         vz[id]+=convpz[ik];
417     }
418     __global__ void cuda_PML_vx_4(float *p, float *convpx, float *bx, float *vx,
419         float _dx, int npml, int nnz, int nnx)
420     {
421         // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
422             direction
423         // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
424             direction
425         // only 2 blocks used vertically, blockIdx.y=0, 1
426
427         // id: position in whole zone(including PML)
428         // ik: position in top or bottom PML zone itself
429         // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
430         int i1=threadIdx.x+blockIdx.x*blockDim.x;
431         int i2=threadIdx.y+blockIdx.y*npml;
432         int ik=i1+i2*nnz;
433         int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
434
435         __shared__ float s_p[Block_Size1][35]; // npml+3=35; Block_SizeX=32;
436             Block_SizeY=8;
437         s_p[threadIdx.x][threadIdx.y+1]=p[id];
438         if (threadIdx.y<1)
439         {
440             if (blockIdx.y)
441                 s_p[threadIdx.x][threadIdx.y]=p[
442                     id-nnz];
443             else
444                 s_p[threadIdx.x][threadIdx.y
445                     ]=0.0f;
446         }
447         if (threadIdx.y>29)
448         {
449             if (blockIdx.y<gridDim.y-1)
450                 s_p[threadIdx.x][threadIdx.y+3]=
451                     p[id+2*nnz];
452             else
453                 s_p[threadIdx.x][threadIdx.y
454                     +3]=0.0f;
455         }
456         __syncthreads();
457
458         float diff2=1.125f*(s_p[threadIdx.x][threadIdx.y+2]-s_p[threadIdx.x][
459             threadIdx.y+1])
460             -0.0416666666666667f*(s_p[threadIdx.x][threadIdx.y+3]-s_p
461                 [threadIdx.x][threadIdx.y]);
462         convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
463         vx[id]+=convpx[ik];
464     }
465 }
466
467 __global__ void cuda_forward_up_4(float *up, float *vx, float *vz, float _dx,
468     float _dz, int npml, int nnz, int nnx)
469 {
470     int i1=blockIdx.x*blockDim.x+threadIdx.x;
471     int i2=blockIdx.y*blockDim.y+threadIdx.y;
472     int id=i1+i2*nnz;
473
474     __shared__ float s_vx[Block_Size1][Block_Size2+3];
475     __shared__ float s_vz[Block_Size1+3][Block_Size2];

```



```

459     s_vx[threadIdx.x][threadIdx.y+2]=vx[id];
460     s_vz[threadIdx.x+2][threadIdx.y]=vz[id];
461
462     if (threadIdx.x<2)
463     {
464         if (blockIdx.x)                s_vz[threadIdx.x][threadIdx.y]=
            vz[id-2];
465         else                s_vz[threadIdx.x][threadIdx.y]
            =0.0f;
466     }
467     if (threadIdx.x>blockDim.x-2)
468     {
469         if (blockIdx.x<gridDim.x-1)    s_vz[threadIdx.x+3][threadIdx.y]
            =vz[id+1];
470         else                s_vz[threadIdx.x+3][threadIdx.y]
            =0.0f;
471     }
472     if (threadIdx.y<2)
473     {
474         if (blockIdx.y)                s_vx[threadIdx.x][threadIdx.y]=
            vx[id-2*nnz];
475         else                s_vx[threadIdx.x][threadIdx.y]
            =0.0f;
476     }
477     if (threadIdx.y>blockDim.y-2)
478     {
479         if (blockIdx.y<gridDim.y-1)    s_vx[threadIdx.x][threadIdx.y
            +3]=vx[id+nnz];
480         else                s_vx[threadIdx.x][threadIdx.y
            +3]=0.0f;
481     }
482     __syncthreads();
483
484     float diff2=1.125f*(s_vx[threadIdx.x][threadIdx.y+2]-s_vx[threadIdx.x][
            threadIdx.y+1])
485             -0.0416666666666667f*(s_vx[threadIdx.x][threadIdx.y+3]-
            s_vx[threadIdx.x][threadIdx.y]);
486     float diff1=1.125f*(s_vz[threadIdx.x+2][threadIdx.y]-s_vz[threadIdx.x
            +1][threadIdx.y])+
487             -0.0416666666666667f*(s_vz[threadIdx.x+3][threadIdx.y]-
            s_vz[threadIdx.x][threadIdx.y]);
488     up[id]=_dz*diff1+_dx*diff2;
489 }
490
491 __global__ void cuda_PML_upz_4(float *up,    float *convvz,    float *bz,    float *vz,
            float _dz,    int npml,    int nnz,    int nnx)
492 {
493     // bz1: top and bottom PML ABC coefficients , decay p (px,pz) along z
            direction
494     // bz2: top and bottom PML ABC coefficients , decay v (vx,vz) along z
            direction
495     // only 2 blocks used vertically , blockIdx.y=0, 1
496
497     // id: position in whole zone(including PML)
498     // ik: position in top or bottom PML zone itself
499     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
500     int i1=threadIdx.x+blockIdx.x*npml;
501     int i2=threadIdx.y+blockIdx.y*blockDim.y;
502     int ik=i1+2*npml*i2;
503     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
504

```

```

505     __shared__ float s_vz[35][Block_Size2];
506     s_vz[threadIdx.x+2][threadIdx.y]=vz[id];
507     if (threadIdx.x<2)
508     {
509         if (blockIdx.x)                s_vz[threadIdx.x][threadIdx.y]=
            vz[id-2];
510         else                s_vz[threadIdx.x][threadIdx.y]
            =0.0f;
511     }
512     if (threadIdx.x>30)
513     {
514         if (blockIdx.x<gridDim.x-1)    s_vz[threadIdx.x+3][threadIdx.y]
            =vz[id+1];
515         else                s_vz[threadIdx.x+3][threadIdx.y]
            =0.0f;
516     }
517     __syncthreads();
518
519     float diff1=1.125f*(s_vz[threadIdx.x+2][threadIdx.y]-s_vz[threadIdx.x
        +1][threadIdx.y])+
520         -0.04166666666666667f*(s_vz[threadIdx.x+3][threadIdx.y]-
            s_vz[threadIdx.x][threadIdx.y]);
521     convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
522     up[id]+=convvz[ik];
523 }
524 __global__ void cuda_PML_upx_4(float *up, float *convvx, float *bx, float *vx,
    float _dx, int npml, int nnz, int nnx)
525 {
526     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
        direction
527     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
        direction
528     // only 2 blocks used vertically, blockIdx.y=0, 1
529
530     // id: position in whole zone(including PML)
531     // ik: position in top or bottom PML zone itself
532     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
533     int i1=threadIdx.x+blockIdx.x*blockDim.x;
534     int i2=threadIdx.y+blockIdx.y*npml;
535     int ik=i1+i2*nnz;
536     int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
537
538     __shared__ float s_vx[Block_Size1][35];
539     s_vx[threadIdx.x][threadIdx.y+2]=vx[id];
540     if (threadIdx.y<2)
541     {
542         if (blockIdx.y)                s_vx[threadIdx.x][threadIdx.y]=
            vx[id-2*nnz];
543         else                s_vx[threadIdx.x][threadIdx.y]
            =0.0f;
544     }
545     if (threadIdx.y>30)
546     {
547         if (blockIdx.y<gridDim.y-1)    s_vx[threadIdx.x][threadIdx.y]
            +3]=vx[id+nnz];
548         else                s_vx[threadIdx.x][threadIdx.y]
            +3]=0.0f;
549     }
550     __syncthreads();
551

```

```

552         float diff2=1.125f*(s_vx[threadIdx.x][threadIdx.y+2]-s_vx[threadIdx.x][
553             threadIdx.y+1])
554             -0.0416666666666667f*(s_vx[threadIdx.x][threadIdx.y+3]-
555                 s_vx[threadIdx.x][threadIdx.y]);
556         convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
557         up[id]+=convvx[ik];
558     }
559 //===== NJ=6
560 --global-- void cuda_forward_v_6(float *p, float *vx, float *vz, float _dx,
561     float _dz, int npml, int nnz, int nnx)
562 {
563     int i1=blockIdx.x*blockDim.x+threadIdx.x;
564     int i2=blockIdx.y*blockDim.y+threadIdx.y;
565     int id=i1+i2*nnz;
566
567     __shared__ float s_p[Block_Size1+5][Block_Size2+5];
568     s_p[threadIdx.x+2][threadIdx.y+2]=p[id];
569     if (threadIdx.x<2)
570     {
571         if (blockIdx.x)
572             s_p[threadIdx.x][threadIdx.y+2]=
573             p[id-2];
574         else
575             s_p[threadIdx.x][threadIdx.y
576                 +2]=0.0f;
577     }
578     if (threadIdx.x>blockDim.x-4)
579     {
580         if (blockIdx.x<gridDim.x-1)
581             s_p[threadIdx.x+5][threadIdx.y
582                 +2]=p[id+3];
583         else
584             s_p[threadIdx.x+5][threadIdx.y
585                 +2]=0.0f;
586     }
587     if (threadIdx.y<2)
588     {
589         if (blockIdx.y)
590             s_p[threadIdx.x+2][threadIdx.y]=
591             p[id-2*nnz];
592         else
593             s_p[threadIdx.x+2][threadIdx.y
594                 ]=0.0f;
595     }
596     if (threadIdx.y>blockDim.y-4)
597     {
598         if (blockIdx.y<gridDim.y-1)
599             s_p[threadIdx.x+2][threadIdx.y
600                 +5]=p[id+3*nnz];
601         else
602             s_p[threadIdx.x+2][threadIdx.y
603                 +5]=0.0f;
604     }
605     __syncthreads();
606
607     float diff1=1.171875f*(s_p[threadIdx.x+3][threadIdx.y+2]-s_p[threadIdx.x
608         +2][threadIdx.y+2])
609         -0.0651041666666667f*(s_p[threadIdx.x+4][threadIdx.y+2]-
610             s_p[threadIdx.x+1][threadIdx.y+2])
611         +0.0046875f*(s_p[threadIdx.x+5][threadIdx.y+2]-s_p[
612             threadIdx.x][threadIdx.y+2]);
613     float diff2=1.171875f*(s_p[threadIdx.x+2][threadIdx.y+3]-s_p[threadIdx.x
614         +2][threadIdx.y+2])
615         -0.0651041666666667f*(s_p[threadIdx.x+2][threadIdx.y+4]-
616             s_p[threadIdx.x+2][threadIdx.y+1])

```

```

595         +0.0046875f*(s_p[threadIdx.x+2][threadIdx.y+5]-s_p[
                    threadIdx.x+2][threadIdx.y]);
596     vz[id]=_dz*diff1;
597     vx[id]=_dx*diff2;
598 }
599
600 --global__ void cuda_PML_vz_6(float *p, float *convpz, float *bz, float *vz,
        float _dz, int npml, int nnz, int nnx)
601 {
602     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
        direction
603     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
        direction
604     // only 2 blocks used vertically, blockIdx.y=0, 1
605
606     // id: position in whole zone(including PML)
607     // ik: position in top or bottom PML zone itself
608     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
609     int i1=threadIdx.x+blockIdx.x*npml;
610     int i2=threadIdx.y+blockIdx.y*blockDim.y;
611     int ik=i1+2*npml*i2;
612     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
613
614     __shared__ float s_p[37][Block_Size2];
615     s_p[threadIdx.x+2][threadIdx.y]=p[id];
616     if (threadIdx.x<2)
617     {
618         if (blockIdx.x)
619             s_p[threadIdx.x][threadIdx.y]=p[
                id-2];
620         else
621             s_p[threadIdx.x][threadIdx.y]
                =0.0f;
622     }
623     if (threadIdx.x>28)
624     {
625         if (blockIdx.x<gridDim.x-1)
626             s_p[threadIdx.x+5][threadIdx.y]=
                p[id+3];
627         else
628             s_p[threadIdx.x+5][threadIdx.y]
                =0.0f;
629     }
630     __syncthreads();
631
632     float diff1=1.171875f*(s_p[threadIdx.x+3][threadIdx.y]-s_p[threadIdx.x
        +2][threadIdx.y])
633         -0.0651041666666667f*(s_p[threadIdx.x+4][threadIdx.y]-s_p[
        threadIdx.x+1][threadIdx.y])
634         +0.0046875f*(s_p[threadIdx.x+5][threadIdx.y]-s_p[
        threadIdx.x][threadIdx.y]);
635     convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
636     vz[id]+=convpz[ik];
637 }
638
639 --global__ void cuda_PML_vx_6(float *p, float *convpx, float *bx, float *vx,
        float _dx, int npml, int nnz, int nnx)
640 {
641     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
        direction
642     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
        direction
643     // only 2 blocks used vertically, blockIdx.y=0, 1
644
645     // id: position in whole zone(including PML)
646     // ik: position in top or bottom PML zone itself

```

```

642 // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
643 int i1=threadIdx.x+blockIdx.x*blockDim.x;
644 int i2=threadIdx.y+blockIdx.y*npml;
645 int ik=i1+i2*nnz;
646 int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
647
648 __shared__ float s_p[Block.Size1][37];
649 s_p[threadIdx.x][threadIdx.y+2]=p[id];
650 if (threadIdx.y<2)
651 {
652     if (blockIdx.y) s_p[threadIdx.x][threadIdx.y]=p[
        id-2*nnz];
653     else s_p[threadIdx.x][threadIdx.y
        ]=0.0f;
654 }
655 if (threadIdx.y>28)
656 {
657     if (blockIdx.y<gridDim.y-1) s_p[threadIdx.x][threadIdx.y+5]=
        p[id+3*nnz];
658     else s_p[threadIdx.x][threadIdx.y
        +5]=0.0f;
659 }
660 __syncthreads();
661
662 float diff2=1.171875f*(s_p[threadIdx.x][threadIdx.y+3]-s_p[threadIdx.x][
        threadIdx.y+2])
663         -0.065104166666667f*(s_p[threadIdx.x][threadIdx.y+4]-s_p
        [threadIdx.x][threadIdx.y+1])
664         +0.0046875f*(s_p[threadIdx.x][threadIdx.y+5]-s_p[
        threadIdx.x][threadIdx.y]);
665 convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
666 vx[id]+=convpx[ik];
667 }
668 __global__ void cuda_forward_up_6(float *up, float *vx, float *vz, float _dx,
        float _dz, int npml, int nnz, int nnx)
669 {
670     int i1=blockIdx.x*blockDim.x+threadIdx.x;
671     int i2=blockIdx.y*blockDim.y+threadIdx.y;
672     int id=i1+i2*nnz;
673
674     __shared__ float s_vx[Block.Size1][Block.Size2+5];
675     __shared__ float s_vz[Block.Size1+5][Block.Size2];
676     s_vx[threadIdx.x][threadIdx.y+3]=vx[id];
677     s_vz[threadIdx.x+3][threadIdx.y]=vz[id];
678
679     if (threadIdx.x<3)
680     {
681         if (blockIdx.x) s_vz[threadIdx.x][threadIdx.y]=
            vz[id-3];
682         else s_vz[threadIdx.x][threadIdx.y
            ]=0.0f;
683     }
684     if (threadIdx.x>blockDim.x-3)
685     {
686         if (blockIdx.x<gridDim.x-1) s_vz[threadIdx.x+5][threadIdx.y
            ]=vz[id+2];
687         else s_vz[threadIdx.x+5][threadIdx.y
            ]=0.0f;
688     }
689     if (threadIdx.y<3)
690     {

```

```

691         if (blockIdx.y)                                s_vx[threadIdx.x][threadIdx.y]=
            vx[id-3*nnz];
692         else                                            s_vx[threadIdx.x][threadIdx.y]
            ]=0.0f;
693     }
694     if (threadIdx.y>blockDim.y-3)
695     {
696         if (blockIdx.y<gridDim.y-1)                    s_vx[threadIdx.x][threadIdx.y
            +5]=vx[id+2*nnz];
697         else                                            s_vx[threadIdx.x][threadIdx.y
            +5]=0.0f;
698     }
699     __syncthreads();
700
701     float diff2=1.171875f*(s_vx[threadIdx.x][threadIdx.y+3]-s_vx[threadIdx.x
702         ][threadIdx.y+2])
703         -0.0651041666666667f*(s_vx[threadIdx.x][threadIdx.y+4]-
            s_vx[threadIdx.x][threadIdx.y+1])+
704         0.0046875f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
            threadIdx.x][threadIdx.y]);
705     float diff1=1.171875f*(s_vz[threadIdx.x+3][threadIdx.y]-s_vz[threadIdx.x
706         +2][threadIdx.y])+
707         -0.0651041666666667f*(s_vz[threadIdx.x+4][threadIdx.y]-
            s_vz[threadIdx.x+1][threadIdx.y])+
708         0.0046875f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
            threadIdx.x][threadIdx.y]);
709     up[id]=(_dz*diff1+_dx*diff2);
710 }
711
712 --global-- void cuda_PML_upz_6(float *up, float *convvz, float *bz, float *vz,
713     float _dz, int npml, int nnz, int nnx)
714 {
715     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
716     // direction
717     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
718     // direction
719     // only 2 blocks used vertically, blockIdx.y=0, 1
720
721     // id: position in whole zone(including PML)
722     // ik: position in top or bottom PML zone itself
723     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
724     int i1=threadIdx.x+blockIdx.x*npml;
725     int i2=threadIdx.y+blockIdx.y*blockDim.y;
726     int ik=i1+2*npml*i2;
727     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
728
729     __shared__ float s_vz[37][Block_Size2]; // npml+5=37; Block_SizeX=32;
730     Block_SizeY=8;
731     s_vz[threadIdx.x+3][threadIdx.y]=vz[id];
732     if (threadIdx.x<3)
733     {
734         if (blockIdx.x)                                s_vz[threadIdx.x][threadIdx.y]=
            vz[id-3];
735         else                                            s_vz[threadIdx.x][threadIdx.y]
            ]=0.0f;
736     }
737     if (threadIdx.x>29)
738     {
739         if (blockIdx.x<gridDim.x-1)                    s_vz[threadIdx.x+5][threadIdx.y
            ]=vz[id+2];
740     }

```

```

735         else                                     s_vz[threadIdx.x+5][threadIdx.y
736             ]=0.0f;
737     }
738     __syncthreads();
739     float diff1=1.171875f*(s_vz[threadIdx.x+3][threadIdx.y]-s_vz[threadIdx.x
740         +2][threadIdx.y])
741         -0.0651041666666667f*(s_vz[threadIdx.x+4][threadIdx.y]-
742             s_vz[threadIdx.x+1][threadIdx.y])
743         +0.0046875f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
744             threadIdx.x][threadIdx.y]);
745     convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
746     up[id]+=convvz[ik];
747 }
748 --global-- void cuda_PML_upx_6(float *up, float *convvx, float *bx, float *vx,
749     float _dx, int npml, int nnz, int nnx)
750 {
751     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
752     // direction
753     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
754     // direction
755     // only 2 blocks used vertically, blockIdx.y=0, 1
756
757     // id: position in whole zone(including PML)
758     // ik: position in top or bottom PML zone itself
759     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
760     int i1=threadIdx.x+blockIdx.x*blockDim.x;
761     int i2=threadIdx.y+blockIdx.y*npml;
762     int ik=i1+i2*nnz;
763     int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
764
765     __shared__ float s_vx[Block_Size1][37];
766     s_vx[threadIdx.x][threadIdx.y+3]=vx[id];
767     if (threadIdx.y<3)
768     {
769         if (blockIdx.y)                             s_vx[threadIdx.x][threadIdx.y]=
770             vx[id-3*nnz];
771         else
772             s_vx[threadIdx.x][threadIdx.y
773                 ]=0.0f;
774     }
775     if (threadIdx.y>29)
776     {
777         if (blockIdx.y<gridDim.y-1)                 s_vx[threadIdx.x][threadIdx.y
778             +5]=vx[id+2*nnz];
779         else
780             s_vx[threadIdx.x][threadIdx.y
781                 +5]=0.0f;
782     }
783     __syncthreads();
784
785     float diff2=1.171875f*(s_vx[threadIdx.x][threadIdx.y+3]-s_vx[threadIdx.x
786         ][threadIdx.y+2])
787         -0.0651041666666667f*(s_vx[threadIdx.x][threadIdx.y+4]-
788             s_vx[threadIdx.x][threadIdx.y+1])
789         +0.0046875f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
790             threadIdx.x][threadIdx.y]);
791     convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
792     up[id]+=convvx[ik];
793 }
794
795
796

```

```

781 //===== NJ=8
782 //=====
783 --global-- void cuda_forward_v_8(float *p, float *vx, float *vz, float _dx,
784     float _dz, int npml, int nnz, int nnx)
785 {
786     int i1=blockIdx.x*blockDim.x+threadIdx.x;
787     int i2=blockIdx.y*blockDim.y+threadIdx.y;
788     int id=i1+nnz*i2;
789
790     __shared__ float s_p[Block_Size1+7][Block_Size2+7];
791     s_p[threadIdx.x+3][threadIdx.y+3]=p[id];
792     if (threadIdx.x<3)
793     {
794         if (blockIdx.x) s_p[threadIdx.x][threadIdx.y+3]=
795             p[id-3];
796         else s_p[threadIdx.x][threadIdx.y
797             +3]=0.0f;
798     }
799     if (threadIdx.x>blockDim.x-5)
800     {
801         if (blockIdx.x<gridDim.x-1) s_p[threadIdx.x+7][threadIdx.y
802             +3]=p[id+4];
803         else s_p[threadIdx.x+7][threadIdx.y
804             +3]=0.0f;
805     }
806     if (threadIdx.y<3)
807     {
808         if (blockIdx.y) s_p[threadIdx.x+3][threadIdx.y]=
809             p[id-3*nnz];
810         else s_p[threadIdx.x+3][threadIdx.y
811             ]=0.0f;
812     }
813     if (threadIdx.y>blockDim.y-5)
814     {
815         if (blockIdx.y<gridDim.y-1) s_p[threadIdx.x+3][threadIdx.y
816             +7]=p[id+4*nnz];
817         else s_p[threadIdx.x+3][threadIdx.y
818             +7]=0.0f;
819     }
820     __syncthreads();
821
822     float diff1=1.1962890625000f*(s_p[threadIdx.x+4][threadIdx.y+3]-s_p[
823         threadIdx.x+3][threadIdx.y+3])
824         -0.0797526041667f*(s_p[threadIdx.x+5][threadIdx.y+3]-s_p[
825             threadIdx.x+2][threadIdx.y+3])
826         +0.0095703125000f*(s_p[threadIdx.x+6][threadIdx.y+3]-s_p[
827             threadIdx.x+1][threadIdx.y+3])
828         -0.0006975446429f*(s_p[threadIdx.x+7][threadIdx.y+3]-s_p[
829             threadIdx.x][threadIdx.y+3]);
830     float diff2=1.1962890625000f*(s_p[threadIdx.x+3][threadIdx.y+4]-s_p[
831         threadIdx.x+3][threadIdx.y+3])
832         -0.0797526041667f*(s_p[threadIdx.x+3][threadIdx.y+5]-s_p[
833             threadIdx.x+3][threadIdx.y+2])
834         +0.0095703125000f*(s_p[threadIdx.x+3][threadIdx.y+6]-s_p[
835             threadIdx.x+3][threadIdx.y+1])
836         -0.0006975446429f*(s_p[threadIdx.x+3][threadIdx.y+7]-s_p[
837             threadIdx.x+3][threadIdx.y]);
838
839     vz[id]=_dz*diff1;
840     vx[id]=_dx*diff2;
841 }

```



```

824
825
826 --global__ void cuda_PML_vz_8(float *p, float *convpz, float *bz, float *vz,
      float _dz, int npml, int nnz, int nnx)
827 {
828     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
      direction
829     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
      direction
830     // only 2 blocks used vertically, blockIdx.y=0, 1
831
832     // id: position in whole zone(including PML)
833     // ik: position in top or bottom PML zone itself
834     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
835     int i1=threadIdx.x+blockIdx.x*npml;
836     int i2=threadIdx.y+blockIdx.y*blockDim.y;
837     int ik=i1+2*npml*i2;
838     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
839
840     __shared__ float s_p[39][Block_Size2];
841     s_p[threadIdx.x+3][threadIdx.y]=p[id];
842     if (threadIdx.x<3)
843     {
844         if (blockIdx.x)                s_p[threadIdx.x][threadIdx.y]=p[
      id-3];
845         else                            s_p[threadIdx.x][threadIdx.y
      ]=0.0f;
846     }
847     if (threadIdx.x>27)
848     {
849         if (blockIdx.x<gridDim.x-1)    s_p[threadIdx.x+7][threadIdx.y]=
      p[id+4];
850         else                            s_p[threadIdx.x+7][threadIdx.y
      ]=0.0f;
851     }
852     __syncthreads();
853
854     float diff1=1.1962890625000f*(s_p[threadIdx.x+4][threadIdx.y]-s_p[
      threadIdx.x+3][threadIdx.y])
855         -0.0797526041667f*(s_p[threadIdx.x+5][threadIdx.y]-s_p[
      threadIdx.x+2][threadIdx.y])
856         +0.0095703125000f*(s_p[threadIdx.x+6][threadIdx.y]-s_p[
      threadIdx.x+1][threadIdx.y])
857         -0.0006975446429f*(s_p[threadIdx.x+7][threadIdx.y]-s_p[
      threadIdx.x][threadIdx.y]);
858     convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
859     vz[id]+=convpz[ik];
860 }
861 --global__ void cuda_PML_vx_8(float *p, float *convpx, float *bx, float *vx,
      float _dx, int npml, int nnz, int nnx)
862 {
863     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
      direction
864     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
      direction
865     // only 2 blocks used vertically, blockIdx.y=0, 1
866
867     // id: position in whole zone(including PML)
868     // ik: position in top or bottom PML zone itself
869     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
870     int i1=threadIdx.x+blockIdx.x*blockDim.x;

```

```

871     int i2=threadIdx.y+blockIdx.y*npml;
872     int ik=i1+i2*nnz;
873     int id=i1+nnz*(blockIdx.y*(nnx*npml)+threadIdx.y);
874
875     __shared__ float s_p[Block_Size1][39]; // npml+7=39; Block_SizeX=32;
876     Block_SizeY=8;
877     s_p[threadIdx.x][threadIdx.y+3]=p[id];
878     if (threadIdx.y<3)
879     {
880         if (blockIdx.y)                s_p[threadIdx.x][threadIdx.y]=p[
881             id-3*nnz];
882         else                            s_p[threadIdx.x][threadIdx.y
883             ]=0.0f;
884     }
885     if (threadIdx.y>27)//32-4
886     {
887         if (blockIdx.y<gridDim.y-1)    s_p[threadIdx.x][threadIdx.y+7]=
888             p[id+4*nnz];
889         else                            s_p[threadIdx.x][threadIdx.y
890             +7]=0.0f;
891     }
892     __syncthreads();
893
894     float diff2=1.1962890625000f*(s_p[threadIdx.x][threadIdx.y+4]-s_p[
895         threadIdx.x][threadIdx.y+3])
896         -0.0797526041667f*(s_p[threadIdx.x][threadIdx.y+5]-s_p[
897             threadIdx.x][threadIdx.y+2])+
898         0.0095703125000f*(s_p[threadIdx.x][threadIdx.y+6]-s_p[
899             threadIdx.x][threadIdx.y+1])+
900         -0.0006975446429f*(s_p[threadIdx.x][threadIdx.y+7]-s_p[
901             threadIdx.x][threadIdx.y]);
902     convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
903     vx[id]+=convpx[ik];
904 }
905
906 __global__ void cuda_forward_up_8(float *up, float *vx, float *vz, float _dx,
907     float _dz, int npml, int nnz, int nnx)
908 {
909     int i1=blockIdx.x*blockDim.x+threadIdx.x;
910     int i2=blockIdx.y*blockDim.y+threadIdx.y;
911     int id=i1+i2*nnz;
912
913     __shared__ float s_vx[Block_Size1][Block_Size2+7];
914     __shared__ float s_vz[Block_Size1+7][Block_Size2];
915     s_vx[threadIdx.x][threadIdx.y+4]=vx[id];
916     s_vz[threadIdx.x+4][threadIdx.y]=vz[id];
917
918     if (threadIdx.x<4)
919     {
920         if (blockIdx.x)                s_vz[threadIdx.x][threadIdx.y]=
921             vz[id-4];
922         else                            s_vz[threadIdx.x][threadIdx.y
923             ]=0.0f;
924     }
925     if (threadIdx.x>blockDim.x-4)
926     {
927         if (blockIdx.x<gridDim.x-1)    s_vz[threadIdx.x+7][threadIdx.y
928             ]=vz[id+3];
929         else                            s_vz[threadIdx.x+7][threadIdx.y
930             ]=0.0f;
931     }

```

```

918     if (threadIdx.y<4)
919     {
920         if (blockIdx.y)                s_vx[threadIdx.x][threadIdx.y]=
            vx[id-4*nnz];
921         else                            s_vx[threadIdx.x][threadIdx.y]
            ]=0.0f;
922     }
923     if (threadIdx.y>blockDim.y-4)
924     {
925         if (blockIdx.y<gridDim.y-1)    s_vx[threadIdx.x][threadIdx.y
            +7]=vx[id+3*nnz];
926         else                            s_vx[threadIdx.x][threadIdx.y
            +7]=0.0f;
927     }
928     __syncthreads();
929
930
931     float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+4]-s_vx[
            threadIdx.x][threadIdx.y+3])
932             -0.0797526041667f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx
            [threadIdx.x][threadIdx.y+2])+
933             0.0095703125000f*(s_vx[threadIdx.x][threadIdx.y+6]-s_vx[
            threadIdx.x][threadIdx.y+1])+
934             -0.0006975446429f*(s_vx[threadIdx.x][threadIdx.y+7]-s_vx
            [threadIdx.x][threadIdx.y]);
935     float diff1=1.1962890625000f*(s_vz[threadIdx.x+4][threadIdx.y]-s_vz[
            threadIdx.x+3][threadIdx.y])+
936             -0.0797526041667f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz
            [threadIdx.x+2][threadIdx.y])+
937             0.0095703125000f*(s_vz[threadIdx.x+6][threadIdx.y]-s_vz[
            threadIdx.x+1][threadIdx.y])+
938             -0.0006975446429f*(s_vz[threadIdx.x+7][threadIdx.y]-s_vz
            [threadIdx.x][threadIdx.y]);
939     up[id]=_dz*diff1+_dx*diff2;
940 }
941 __global__ void cuda_PML_upz_8(float *up, float *convvz, float *bz, float *vz,
            float _dz, int npml, int nnz, int nnx)
942 {
943     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
944     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
945     // only 2 blocks used vertically, blockIdx.y=0, 1
946
947     // id: position in whole zone(including PML)
948     // ik: position in top or bottom PML zone itself
949     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
950     int i1=threadIdx.x+blockIdx.x*npml;
951     int i2=threadIdx.y+blockIdx.y*blockDim.y;
952     int ik=i1+2*npml*i2;
953     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
954
955     __shared__ float s_vz[39][Block_Size2];
956     s_vz[threadIdx.x+4][threadIdx.y]=vz[id];
957     if (threadIdx.x<4)
958     {
959         if (blockIdx.x)                s_vz[threadIdx.x][threadIdx.y]=
            vz[id-4];
960         else                            s_vz[threadIdx.x][threadIdx.y]
            ]=0.0f;
961     }

```

```

962     if (threadIdx.x>28)
963     {
964         if (blockIdx.x<gridDim.x-1)      s_vz[threadIdx.x+7][threadIdx.y
965         ]=vz[id+3];
966         else                               s_vz[threadIdx.x+7][threadIdx.y
967         ]=0.0f;
968     }
969     __syncthreads();
970
971     float diff1=1.1962890625000f*(s_vz[threadIdx.x+4][threadIdx.y]-s_vz[
972     threadIdx.x+3][threadIdx.y])
973     -0.0797526041667f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
974     threadIdx.x+2][threadIdx.y])
975     +0.0095703125000f*(s_vz[threadIdx.x+6][threadIdx.y]-s_vz[
976     threadIdx.x+1][threadIdx.y])
977     -0.0006975446429f*(s_vz[threadIdx.x+7][threadIdx.y]-s_vz[
978     threadIdx.x][threadIdx.y]);
979     convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
980     up[id]+=convvz[ik];
981 }
982
983 --global-- void cuda_PML_upx_8(float *up, float *convvx, float *bx, float *vx,
984 float _dx, int npml, int nnz, int nnx)
985 {
986     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
987     // direction
988     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
989     // direction
990     // only 2 blocks used vertically, blockIdx.y=0, 1
991
992     // id: position in whole zone(including PML)
993     // ik: position in top or bottom PML zone itself
994     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
995     int i1=threadIdx.x+blockIdx.x*blockDim.x;
996     int i2=threadIdx.y+blockIdx.y*npml;
997     int ik=i1+i2*nnz;
998     int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
999
1000     __shared__ float s_vx[Block_Size1][39];
1001     s_vx[threadIdx.x][threadIdx.y+4]=vx[id];
1002     if (threadIdx.y<4)
1003     {
1004         if (blockIdx.y)
1005             s_vx[threadIdx.x][threadIdx.y]=
1006             vx[id-4*nnz];
1007         else
1008             s_vx[threadIdx.x][threadIdx.y
1009             ]=0.0f;
1010     }
1011     if (threadIdx.y>28)
1012     {
1013         if (blockIdx.y<gridDim.y-1)      s_vx[threadIdx.x][threadIdx.y
1014         +7]=vx[id+3*nnz];
1015         else                               s_vx[threadIdx.x][threadIdx.y
1016         +7]=0.0f;
1017     }
1018     __syncthreads();
1019
1020     float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+4]-s_vx[
1021     threadIdx.x][threadIdx.y+3])
1022     -0.0797526041667f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
1023     threadIdx.x][threadIdx.y+2])
1024     +0.0095703125000f*(s_vx[threadIdx.x][threadIdx.y+6]-s_vx[
1025     threadIdx.x][threadIdx.y+1])

```

```

1007         -0.0006975446429f*(s_vx[threadIdx.x][threadIdx.y+7]-s_vx
1008             [threadIdx.x][threadIdx.y]);
1009     convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
1010     up[id]+=convvx[ik];
1011 }
1012
1013 //===== NJ=10
1014 //=====
1015 --global-- void cuda_forward_v_10(float *p, float *vx, float *vz, float _dx,
1016     float _dz, int npml, int nnz, int nnx)
1017 {
1018     int i1=blockIdx.x*blockDim.x+threadIdx.x;
1019     int i2=blockIdx.y*blockDim.y+threadIdx.y;
1020     int id=i1+i2*nnz;
1021
1022     __shared__ float s_p[Block_Size1+9][Block_Size2+9];
1023     s_p[threadIdx.x+4][threadIdx.y+4]=p[id];
1024     if (threadIdx.x<4)
1025     {
1026         if (blockIdx.x) s_p[threadIdx.x][threadIdx.y+4]=
1027             p[id-4];
1028         else s_p[threadIdx.x][threadIdx.y
1029             +4]=0.0f;
1030     }
1031     if (threadIdx.x>blockDim.x-6)
1032     {
1033         if (blockIdx.x<gridDim.x-1) s_p[threadIdx.x+9][threadIdx.y
1034             +4]=p[id+5];
1035         else s_p[threadIdx.x+9][threadIdx.y
1036             +4]=0.0f;
1037     }
1038     if (threadIdx.y<4)
1039     {
1040         if (blockIdx.y) s_p[threadIdx.x+4][threadIdx.y]=
1041             p[id-4*nnz];
1042         else s_p[threadIdx.x+4][threadIdx.y
1043             ]=0.0f;
1044     }
1045     if (threadIdx.y>blockDim.y-6)
1046     {
1047         if (blockIdx.y<gridDim.y-1) s_p[threadIdx.x+4][threadIdx.y
1048             +9]=p[id+5*nnz];
1049         else s_p[threadIdx.x+4][threadIdx.y
1050             +9]=0.0f;
1051     }
1052     __syncthreads();
1053
1054     float diff1=1.211242675781250f*(s_p[threadIdx.x+5][threadIdx.y+4]-s_p[
1055         threadIdx.x+4][threadIdx.y+4])
1056         -0.089721679687500f*(s_p[threadIdx.x+6][threadIdx.y+4]-
1057             s_p[threadIdx.x+3][threadIdx.y+4])
1058         +0.013842773437500f*(s_p[threadIdx.x+7][threadIdx.y+4]-
1059             s_p[threadIdx.x+2][threadIdx.y+4])
1060         -0.001765659877232f*(s_p[threadIdx.x+8][threadIdx.y+4]-
1061             s_p[threadIdx.x+1][threadIdx.y+4])
1062         +0.000118679470486f*(s_p[threadIdx.x+9][threadIdx.y+4]-
1063             s_p[threadIdx.x][threadIdx.y+4]);
1064     float diff2= 1.211242675781250f*(s_p[threadIdx.x+4][threadIdx.y+5]-s_p[
1065         threadIdx.x+4][threadIdx.y+4])

```

```

1050         -0.089721679687500f*(s_p[threadIdx.x+4][threadIdx.y+6]-
1051         s_p[threadIdx.x+4][threadIdx.y+3])
1052         +0.013842773437500f*(s_p[threadIdx.x+4][threadIdx.y+7]-
1053         s_p[threadIdx.x+4][threadIdx.y+2])
1054         -0.001765659877232f*(s_p[threadIdx.x+4][threadIdx.y+8]-
1055         s_p[threadIdx.x+4][threadIdx.y+1])
1056         +0.000118679470486f*(s_p[threadIdx.x+4][threadIdx.y+9]-
1057         s_p[threadIdx.x+4][threadIdx.y]);
1058     vz[id]=_dz*diff1;
1059     vx[id]=_dx*diff2;
1060 }
1061
1062 --global-- void cuda_PML_vz_10(float *p, float *convpz, float *bz, float *vz,
1063     float _dz, int npml, int nnz, int nnx)
1064 {
1065     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
1066     // direction
1067     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
1068     // direction
1069     // only 2 blocks used vertically, blockIdx.y=0, 1
1070
1071     // id: position in whole zone(including PML)
1072     // ik: position in top or bottom PML zone itself
1073     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
1074     int i1=threadIdx.x+blockIdx.x*npml;
1075     int i2=threadIdx.y+blockIdx.y*blockDim.y;
1076     int ik=i1+2*npml*i2;
1077     int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;
1078
1079     __shared__ float s_p[41][Block_Size2];
1080     s_p[threadIdx.x+4][threadIdx.y]=p[id];
1081     if (threadIdx.x<4)
1082     {
1083         if (blockIdx.x)
1084             s_p[threadIdx.x][threadIdx.y]=p[
1085                 id-4];
1086         else
1087             s_p[threadIdx.x][threadIdx.y]
1088                 =0.0f;
1089     }
1090     if (threadIdx.x>26)
1091     {
1092         if (blockIdx.x<gridDim.x-1)
1093             s_p[threadIdx.x+9][threadIdx.y]=
1094                 p[id+5];
1095         else
1096             s_p[threadIdx.x+9][threadIdx.y]
1097                 =0.0f;
1098     }
1099     __syncthreads();
1100
1101     float diff1=1.1962890625000f*(s_p[threadIdx.x+5][threadIdx.y]-s_p[
1102         threadIdx.x+4][threadIdx.y])
1103         -0.089721679687500f*(s_p[threadIdx.x+6][threadIdx.y]-s_p[
1104         threadIdx.x+3][threadIdx.y])
1105         +0.013842773437500f*(s_p[threadIdx.x+7][threadIdx.y]-s_p[
1106         threadIdx.x+2][threadIdx.y])
1107         -0.001765659877232f*(s_p[threadIdx.x+8][threadIdx.y]-s_p[
1108         threadIdx.x+1][threadIdx.y])
1109         +0.000118679470486f*(s_p[threadIdx.x+9][threadIdx.y]-s_p[
1110         threadIdx.x][threadIdx.y]);
1111     convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
1112     vz[id]+=convpz[ik];
1113 }

```

```

1094 --global__ void cuda_PML_vx_10(float *p, float *convpx, float *bx, float *vx,
1095     float _dx, int npml, int nnz, int nnx)
1096 {
1097     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
1098     // direction
1099     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
1100     // direction
1101     // only 2 blocks used vertically, blockIdx.y=0, 1
1102
1103     // id: position in whole zone(including PML)
1104     // ik: position in top or bottom PML zone itself
1105     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
1106     int i1=threadIdx.x+blockIdx.x*blockDim.x;
1107     int i2=threadIdx.y+blockIdx.y*npml;
1108     int ik=i1+i2*nnz;
1109     int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
1110     __shared__ float s_p[Block_Size1][41];
1111     s_p[threadIdx.x][threadIdx.y+4]=p[id];
1112     if (threadIdx.y<4)
1113     {
1114         if (blockIdx.y)
1115             s_p[threadIdx.x][threadIdx.y]=p[
1116                 id-4*nnz];
1117         else
1118             s_p[threadIdx.x][threadIdx.y
1119                 ]=0.0f;
1120     }
1121     if (threadIdx.y>26)
1122     {
1123         if (blockIdx.y<gridDim.y-1)
1124             s_p[threadIdx.x][threadIdx.y+9]=
1125                 p[id+5*nnz];
1126         else
1127             s_p[threadIdx.x][threadIdx.y
1128                 +9]=0.0f;
1129     }
1130     __syncthreads();
1131
1132     float diff2=1.1962890625000f*(s_p[threadIdx.x][threadIdx.y+5]-s_p[
1133         threadIdx.x][threadIdx.y+4])
1134         -0.089721679687500f*(s_p[threadIdx.x][threadIdx.y+6]-s_p[
1135             threadIdx.x][threadIdx.y+3])
1136         +0.013842773437500f*(s_p[threadIdx.x][threadIdx.y+7]-s_p[
1137             threadIdx.x][threadIdx.y+2])
1138         -0.001765659877232f*(s_p[threadIdx.x][threadIdx.y+8]-s_p[
1139             threadIdx.x][threadIdx.y+1])
1140         +0.000118679470486f*(s_p[threadIdx.x][threadIdx.y+9]-s_p[
1141             threadIdx.x][threadIdx.y]);
1142     convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
1143     vx[id]+=convpx[ik];
1144 }
1145
1146 --global__ void cuda_forward_up_10(float *up, float *vx, float *vz, float _dx,
1147     float _dz, int npml, int nnz, int nnx)
1148 {
1149     int i1=blockIdx.x*blockDim.x+threadIdx.x;
1150     int i2=blockIdx.y*blockDim.y+threadIdx.y;
1151     int id=i1+i2*nnz;
1152
1153     __shared__ float s_vx[Block_Size1][Block_Size2+9];
1154     __shared__ float s_vz[Block_Size1+9][Block_Size2];
1155     s_vx[threadIdx.x][threadIdx.y+5]=vx[id];
1156     s_vz[threadIdx.x+5][threadIdx.y]=vz[id];
1157
1158     if (threadIdx.x<5)

```



```

1142     {
1143         if (blockIdx.x)                s_vz[threadIdx.x][threadIdx.y]=
            vz[id-5];
1144         else                            s_vz[threadIdx.x][threadIdx.y]
            =0.0f;
1145     }
1146     if (threadIdx.x>blockDim.x-5)
1147     {
1148         if (blockIdx.x<gridDim.x-1)    s_vz[threadIdx.x+9][threadIdx.y]
            =vz[id+4];
1149         else                            s_vz[threadIdx.x+9][threadIdx.y]
            =0.0f;
1150     }
1151     if (threadIdx.y<5)
1152     {
1153         if (blockIdx.y)                s_vx[threadIdx.x][threadIdx.y]=
            vx[id-5*nnz];
1154         else                            s_vx[threadIdx.x][threadIdx.y]
            =0.0f;
1155     }
1156     if (threadIdx.y>blockDim.y-5)
1157     {
1158         if (blockIdx.y<gridDim.y-1)    s_vx[threadIdx.x][threadIdx.y]
            +9=vx[id+4*nnz];
1159         else                            s_vx[threadIdx.x][threadIdx.y]
            +9=0.0f;
1160     }
1161     __syncthreads();
1162
1163     float diff1=1.1962890625000f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
        threadIdx.x+4][threadIdx.y])
1164         -0.089721679687500f*(s_vz[threadIdx.x+6][threadIdx.y]-
            s_vz[threadIdx.x+3][threadIdx.y])
1165         +0.013842773437500f*(s_vz[threadIdx.x+7][threadIdx.y]-
            s_vz[threadIdx.x+2][threadIdx.y])
1166         -0.001765659877232f*(s_vz[threadIdx.x+8][threadIdx.y]-
            s_vz[threadIdx.x+1][threadIdx.y])
1167         +0.000118679470486f*(s_vz[threadIdx.x+9][threadIdx.y]-
            s_vz[threadIdx.x][threadIdx.y]);
1168     float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
        threadIdx.x][threadIdx.y+4])
1169         -0.089721679687500f*(s_vx[threadIdx.x][threadIdx.y+6]-
            s_vx[threadIdx.x][threadIdx.y+3])
1170         +0.013842773437500f*(s_vx[threadIdx.x][threadIdx.y+7]-
            s_vx[threadIdx.x][threadIdx.y+2])
1171         -0.001765659877232f*(s_vx[threadIdx.x][threadIdx.y+8]-
            s_vx[threadIdx.x][threadIdx.y+1])
1172         +0.000118679470486f*(s_vx[threadIdx.x][threadIdx.y+9]-
            s_vx[threadIdx.x][threadIdx.y]);
1173     up[id]=(_dz*diff1+_dx*diff2);
1174 }
1175 __global__ void cuda_PML_upz_10(float *up, float *convvz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
1176 {
1177     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
        direction
1178     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
        direction
1179     // only 2 blocks used vertically, blockIdx.y=0, 1
1180
1181     // id: position in whole zone(including PML)

```



```

1182 // ik: position in top or bottom PML zone itself
1183 // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
1184 int i1=threadIdx.x+blockIdx.x*npml;
1185 int i2=threadIdx.y+blockIdx.y*blockDim.y;
1186 int ik=i1+2*npml*i2;
1187 int id=blockIdx.x*(nnz*npml)+threadIdx.x+nnz*i2;
1188
1189 __shared__ float s_vz[41][Block_Size2];
1190 s_vz[threadIdx.x+5][threadIdx.y]=vz[id];
1191 if (threadIdx.x<5)
1192 {
1193     if (blockIdx.x) s_vz[threadIdx.x][threadIdx.y]=
1194         vz[id-5];
1195     else s_vz[threadIdx.x][threadIdx.y]
1196         =0.0f;
1197 }
1198 if (threadIdx.x>27)
1199 {
1200     if (blockIdx.x<gridDim.x-1) s_vz[threadIdx.x+9][threadIdx.y]
1201         =vz[id+4];
1202     else s_vz[threadIdx.x+9][threadIdx.y]
1203         =0.0f;
1204 }
1205 __syncthreads();
1206
1207 float diff1=1.1962890625000f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
1208     threadIdx.x+4][threadIdx.y])
1209     -0.089721679687500f*(s_vz[threadIdx.x+6][threadIdx.y]-
1210         s_vz[threadIdx.x+3][threadIdx.y])
1211     +0.013842773437500f*(s_vz[threadIdx.x+7][threadIdx.y]-
1212         s_vz[threadIdx.x+2][threadIdx.y])
1213     -0.001765659877232f*(s_vz[threadIdx.x+8][threadIdx.y]-
1214         s_vz[threadIdx.x+1][threadIdx.y])
1215     +0.000118679470486f*(s_vz[threadIdx.x+9][threadIdx.y]-
1216         s_vz[threadIdx.x][threadIdx.y]);
1217 convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
1218 up[id]+=convvz[ik];
1219 }
1220
1221 __global__ void cuda_PML_upx_10(float *up, float *convvx, float *bx, float *vx,
1222     float _dx, int npml, int nnz, int nnx)
1223 {
1224     // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
1225     // direction
1226     // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
1227     // direction
1228     // only 2 blocks used vertically, blockIdx.y=0, 1
1229
1230     // id: position in whole zone(including PML)
1231     // ik: position in top or bottom PML zone itself
1232     // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
1233     int i1=threadIdx.x+blockIdx.x*blockDim.x;
1234     int i2=threadIdx.y+blockIdx.y*npml;
1235     int ik=i1+i2*nnz;
1236     int id=i1+nnz*(blockIdx.y*(nnx*npml)+threadIdx.y);
1237
1238     __shared__ float s_vx[Block_Size1][41];
1239     s_vx[threadIdx.x][threadIdx.y+5]=vx[id];
1240     if (threadIdx.y<5)
1241     {

```

```

1230         if (blockIdx.y)                                s_vx[threadIdx.x][threadIdx.y]=
                vx[id-5*nnz];
1231         else                                            s_vx[threadIdx.x][threadIdx.y]
                ]=0.0f;
1232     }
1233     if (threadIdx.y>27)
1234     {
1235         if (blockIdx.y<gridDim.y-1)                    s_vx[threadIdx.x][threadIdx.y
                +9]=vx[id+4*nnz];
1236         else                                            s_vx[threadIdx.x][threadIdx.y
                +9]=0.0f;
1237     }
1238     __syncthreads();
1239
1240     float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
        threadIdx.x][threadIdx.y+4])
1241             -0.089721679687500f*(s_vx[threadIdx.x][threadIdx.y+6]-
                s_vx[threadIdx.x][threadIdx.y+3])+
1242             0.013842773437500f*(s_vx[threadIdx.x][threadIdx.y+7]-
                s_vx[threadIdx.x][threadIdx.y+2])
1243             -0.001765659877232f*(s_vx[threadIdx.x][threadIdx.y+8]-
                s_vx[threadIdx.x][threadIdx.y+1])+
1244             0.000118679470486f*(s_vx[threadIdx.x][threadIdx.y+9]-
                s_vx[threadIdx.x][threadIdx.y]);
1245     convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
1246     up[id]+=convvx[ik];
1247 }
1248
1249 // update wavefield p
1250 // if frsf==true, free surface boundary condition
1251 __global__ void cuda_step_forward(float *vel, float *up, float *p0, float *p1,
        float dt, bool frsf, int npml, int nnz, int nnx)
1252 {
1253     int i1=blockIdx.x*blockDim.x+threadIdx.x;
1254     int i2=blockIdx.y*blockDim.y+threadIdx.y;
1255     int id=i1+i2*nnz;
1256     float c=dt*vel[id];c=c*c;
1257     p0[id]=2*p1[id]-p0[id]+c*up[id];
1258
1259     if (frsf && i1<npml)    p0[id]=0.0;
1260 }
1261
1262 //===== read and write/save the boundary
1263 //=====
1264 // read and write the inner computation zone boundary coefficients from and into
    RAM along z direction
1265 // read==false, write/save boundary; read==true, read the boundary
1266 __global__ void cuda_rw_boundaryb(float *boundarytb, float *p, int npml, int
    nnz, int nnx, int NJ, bool read)
1267 {
1268     //int nx=nnx-2*npml;
1269     int nz=nnz-2*npml;
1270     int i1=threadIdx.x+blockDim.x*blockIdx.x;
1271     int i2=threadIdx.y+blockDim.y*blockIdx.y;
1272     int id=i1+2*(NJ-1)*i2;
1273     int i1p=i1+npml-(NJ-1);
1274     int i2p=i2;
1275     int idp=i1p+nnz*i2p;
1276
1277     if (i1<NJ-1 && i2<nnx)

```

```

1278     {
1279         if(read)
1280         {
1281             p[idp]=boundarytb[id];
1282             p[idp+nz+NJ-1]=boundarytb[id+NJ-1];
1283         }
1284         else
1285         {
1286             boundarytb[id]=p[idp];
1287             boundarytb[id+NJ-1]=p[idp+nz+NJ-1];
1288         }
1289     }
1290 }
1291
1292 // read and write the inner computation zone boundary coefficients from and into
    RAM along x direction
1293 // read==false, write and save boundary; read==true, read the boundary
1294 --global-- void cuda_rw_boundarylr(float *boundarylr, float *p, int npml, int
    nnz, int nnx, int NJ, bool read)
1295 {
1296     int nx=nnx-2*npml;
1297     //int nz=nnz-2*npml;
1298     int i1=threadIdx.x+blockDim.x*blockIdx.x;
1299     int i2=threadIdx.y+blockDim.y*blockIdx.y;
1300     int id=i1+nnz*i2;
1301     int i1p=i1;
1302     int i2p=i2+npml-(NJ-1);
1303     int idp=i1p+nnz*i2p;
1304
1305     if (i1<nnz && i2<NJ-1)
1306     {
1307         if (read)
1308         {
1309             p[idp]=boundarylr[id];
1310             p[idp+nnz*(nx+NJ-1)]=boundarylr[id+nnz*(NJ-1)];
1311         }
1312         else
1313         {
1314             boundarylr[id]=p[idp];
1315             boundarylr[id+nnz*(NJ-1)]=p[idp+nnz*(nx+NJ-1)];
1316         }
1317     }
1318 }
1319
1320
1321
1322 //===== imaging condition
    =====
1323 --global-- void cuda_cross_correlate(float *Isg, float *Iss, float *sp, float *
    gp, int npml, int nnz, int nnx)
1324 {
1325     int i1=threadIdx.x+blockDim.x*blockIdx.x;
1326     int i2=threadIdx.y+blockDim.y*blockIdx.y;
1327     int id=i1+i2*nnz;
1328
1329     if(i1>=npml && i1<nnz-npml && i2>=npml && i2<nnx-npml)
1330     {
1331         float ps=sp[id];
1332         float pg=gp[id];
1333         Isg[id] += ps*pg;
1334         Iss[id] += ps*ps;

```

```

1335     }
1336 }
1337
1338
1339 --global__ void cuda_imaging(float *Isg, float *Iss, float *I1, float *I2, int
      npml, int nnz, int nnx)
1340 {
1341     int nz=nnz-2*npml;
1342     int nx=nnx-2*npml;
1343     int i1=threadIdx.x+blockDim.x*blockIdx.x;
1344     int i2=threadIdx.y+blockDim.y*blockIdx.y;
1345     int id=i1+i2*nnz;
1346
1347     if(i1>=npml && i1<nnz-npml && i2>=npml && i2<nnx-npml)
1348     {
1349         I1[id]+=Isg[id]; // correlation imaging condition
1350         I2[id]+=Isg[id]/(Iss[id]+EPS); // normalized image
1351     }
1352     __syncthreads();
1353
1354     if (i2>=0 && i2<npml && i1>=0 && i1<npml) // top left
1355     {
1356         I1[id]=I1[nnz*npml+npml]; I2[id]=I2[nnz*npml+npml]
1357     };
1358     else if (i2>=npml+nx && i2<nnx && i1>=0 && i1<npml) // top right
1359     {
1360         I1[id]=I1[nnz*(npml+nx-1)+npml]; I2[id]=I2[nnz*(npml+nx
1361         -1)+npml];
1362     }
1363     else if (i2>=0 && i2<npml && i1>=npml+nz && i1<nnz) // bottom left
1364     {
1365         I1[id]=I1[nnz*npml+(npml+nz-1)]; I2[id]=I2[nnz*npml+(npml
1366         +nz-1)];
1367     }
1368     else if (i2>=npml+nx && i2<nnx && i1>=npml+nz && i1<nnz) // bottom right
1369     {
1370         I1[id]=I1[nnz*(npml+nx-1)+(npml+nz-1)]; I2[id]=I2[nnz*(npml+nx
1371         -1)+(npml+nz-1)];
1372     }
1373     else if (i2>=npml && i2<npml+nx && i1>=0 && i1<npml) // top
1374     {
1375         I1[id]=I1[nnz*i2+npml]; I2[id]=I2[nnz*i2+npml];
1376     }
1377     else if (i2>=npml && i2<npml+nx && i1>=npml+nz && i1<nnz) // bottom
1378     {
1379         I1[id]=I1[nnz*i2+(npml+nz-1)]; I2[id]=I2[nnz*i2+(npml+
1380         nz-1)];
1381     }
1382     else if (i2>=0 && i2<npml && i1>=npml && i1<npml+nz) // left
1383     {
1384         I1[id]=I1[nnz*npml+i1]; I2[id]=I2[nnz*npml+i1];
1385     }
1386     else if (i2>=npml+nx && i2<nnx && i1>=npml && i1<npml+nz) // right
1387     {
1388         I1[id]=I1[nnz*(npml+nx-1)+i1]; I2[id]=I2[nnz*(npml+nx
1389         -1)+i1];
1390     }
1391 }
1392
1393 --global__ void cuda_laplace_filter(float *Img, float *laplace, float _dz, float
      _dx, int npml, int nnz, int nnx)
1394 {
1395     int i1=threadIdx.x+blockDim.x*blockIdx.x;
1396     int i2=threadIdx.y+blockDim.y*blockIdx.y;
1397     int id=i1+i2*nnz;
1398     float diff1=0.0f;
1399     float diff2=0.0f;
1400     if(i1>=npml && i1<nnz-npml && i2>=npml && i2<nnx-npml)
1401     {
1402         diff1=Img[id+1]-2.0*Img[id]+Img[id-1];
1403         diff2=Img[id+nnz]-2.0*Img[id]+Img[id-nnz];
1404     }
1405     laplace[id]=_dz*_dz*diff1+_dx*_dx*diff2;
1406 }

```

References

- [1] Edip Baysal, Dan D Kosloff, and John WC Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.
- [2] Jean-Pierre Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of computational physics*, 114(2):185–200, 1994.
- [3] Ake Bjorck. *Numerical methods for least squares problems*. Number 51. Society for Industrial and Applied Mathematics, 1996.
- [4] Jon F Claerbout. Toward a unified theory of reflector mapping. *Geophysics*, 36(3):467–481, 1971.
- [5] Francis Collino and Chrysoula Tsogka. Application of the perfectly matched absorbing layer model to the linear elastodynamic problem in anisotropic heterogeneous media. *Geophysics*, 66(1):294–307, 2001.
- [6] Björn Engquist and Andrew Majda. Absorbing boundary conditions for numerical simulation of waves. *Proceedings of the National Academy of Sciences*, 74(5):1765–1766, 1977.
- [7] Antoine Guitton, Alejandro Valenciano, Dimitri Bevc, and Jon Claerbout. Smoothing imaging condition for shot-profile migration. *Geophysics*, 72(3):S149–S154, 2007.
- [8] Jun Ji. Cgg method for robust inversion and its application to velocity-stack inversion. *Geophysics*, 71(4):R59–R67, 2006.
- [9] Dimitri Komatitsch and Roland Martin. An unsplit convolutional perfectly matched layer improved at grazing incidence for the seismic wave equation. *Geophysics*, 72(5):SM155–SM167, 2007.
- [10] A Pica, JP Diet, and A Tarantola. Nonlinear inversion of seismic reflection data in a laterally invariant medium. *Geophysics*, 55(3):284–292, 1990.
- [11] Gerhard Pratt, Changsoo Shin, et al. Gauss–newton and full newton methods in frequency–space seismic waveform inversion. *Geophysical Journal International*, 133(2):341–362, 1998.
- [12] J Alan Roden and Stephen D Gedney. Convolutional pml (cpml): An efficient fdtd implementation of the cfs-pml for arbitrary media. *Microwave and optical technology letters*, 27(5):334–338, 2000.
- [13] Albert Tarantola. Inversion of seismic reflection data in the acoustic approximation. *Geophysics*, 49(8):1259–1266, 1984.
- [14] Albert Tarantola. A strategy for nonlinear elastic inversion of seismic reflection data. *Geophysics*, 51(10):1893–1903, 1986.
- [15] Jean Virieux and Stéphane Operto. An overview of full-waveform inversion in exploration geophysics. *Geophysics*, 74(6):WCC1–WCC26, 2009.
- [16] Kwangjin Yoon and Kurt J Marfurt. Reverse-time migration using the poynting vector. *Exploration Geophysics*, 37(1):102–107, 2006.