# A GPU tour of wave propagation

Pengliang Yang

December 8, 2013

# Contents



# 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

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

- Constitutive law

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

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 quasiacoustic 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

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

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

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

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

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

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

$$f(x+h)=f(x)+\frac{\partial f(x)}{\partial x}h+\frac{1}{2!}\frac{\partial^2 f(x)}{\partial x^2}h^2+\frac{1}{3!}\frac{\partial^3 f(x)}{\partial x^3}h^3+\ldots. \tag{6}$$

A popular example is

$$(1+x)^\alpha=1+\alpha x+\frac{\alpha(\alpha-1)}{2!}x^2+\ldots+\frac{\alpha(\alpha-1)\cdots(\alpha-n+1)}{n!}x^n+\ldots. \tag{7}$$

Here we mainly consider the following expansion formula:

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

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

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

we provide the derivation:

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

The 1st-order Pade expansion is:

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

The 2nd-order Pade expansion is:

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

And the 3rd-order one is:

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

### 1.2 Approximate the wave equation

The innovative work was done by John Claerbout, well-known as $15^\circ$ 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:

$$\frac{\omega^2}{v^2} = k_x^2 + k_z^2. \tag{13}$$

The down-going wave equation in Fourier domain is

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

Using the different order Pade expansions, we have:

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

The corresponding time domain equations are:

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

# 2 Forward modelling

### 2.1 Elastic wave equation

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

$$\begin{cases} \frac{\partial v_x}{\partial t} = \frac{1}{\rho}(\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial x}) \\ \frac{\partial v_z}{\partial t} = \frac{1}{\rho}(\frac{\partial \tau_{zx}}{\partial z} + \frac{\partial \tau_{zz}}{\partial z}) \\ \frac{\partial \tau_{xx}}{\partial t} = c_{11}\frac{\partial v_x}{\partial x} + c_{13}\frac{\partial v_z}{\partial z} \\ \frac{\partial \tau_{zz}}{\partial t} = c_{13}\frac{\partial v_x}{\partial x} + c_{33}\frac{\partial v_z}{\partial z} \\ \frac{\partial \tau_{xz}}{\partial t} = c_{44}\frac{\partial v_x}{\partial x} + c_{44}\frac{\partial v_z}{\partial z} \end{cases} \tag{17}$$

where $\tau_{ij}$ (sometimes $\sigma_{ij}$) 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

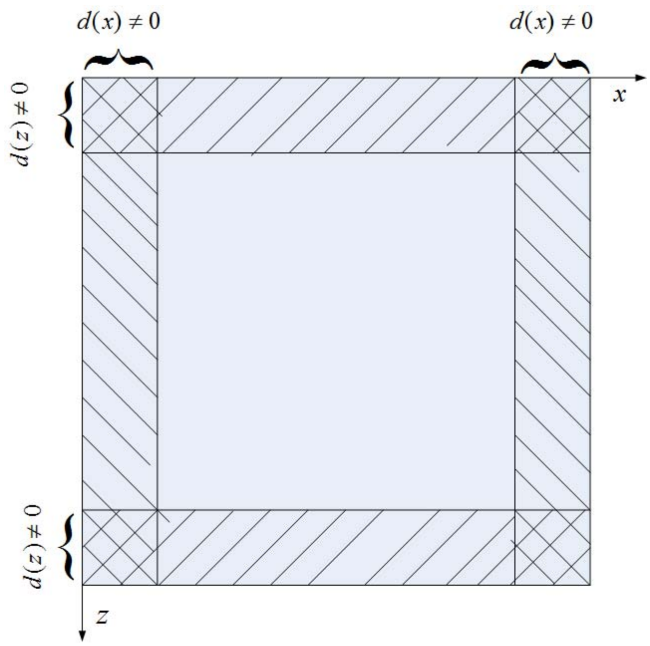


Figure 1: A schematic diagram of PML zone

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]

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

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]:

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

where $L$ indicates the PML thinkness; $x$ 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:

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

where $\Psi_x$, $\Psi_z$ are the convolution terms of $Ax$ and $Az$; $\Phi_x$, $\Phi_z$ are the convolution terms of $Px$ and $Pz$. These convolution terms can be computed via the following relation:

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

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

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

where the auxiliary variables are governed via the following relation

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

## 2.4 Discretization

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

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

It leads to

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

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

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

### 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

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

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

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

Thus, taking first $N$ terms means

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

In matrix form,

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

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

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

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

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

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

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

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

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

The 4th-order dispersion relation is:

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

#### 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 $[ix\Delta x, iz\Delta z]$

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

That is to say,

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

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

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

Thus, we have

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

In summary,

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

If we define:

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

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

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

Then, we have

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

#### 2.4.3 Discretization of NPML

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

$$\frac{\partial f}{\partial t}+df=\gamma. \tag{44}$$

The analytic solution of this equation is

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

In discrete form,

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

Thus,

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

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

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

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

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

For acoustics, we have

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

As can be seen from Eq. (22), we only need to subtract the reflection part $\Omega$ and $\Psi$ after global updating (Eq. (17)). We summarize this precedure as follows:

step 1: perform

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

Step 2: In PML zone, subtract decaying parts:

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

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

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

and

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

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

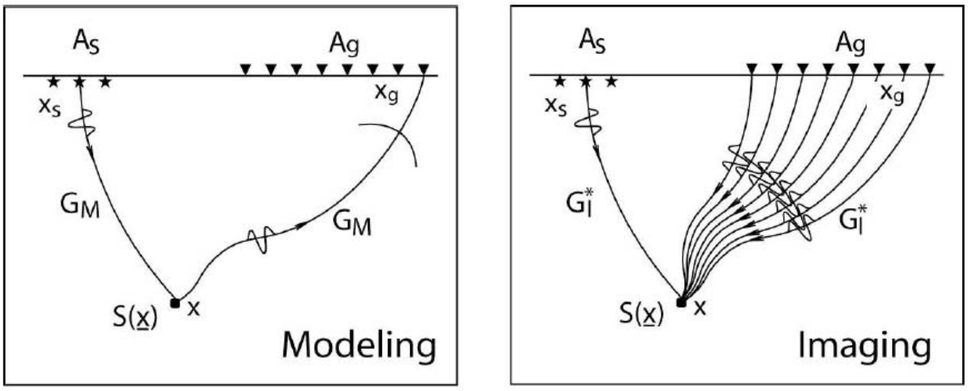


Figure 2: Modeling and migration (imaging)

# 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

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

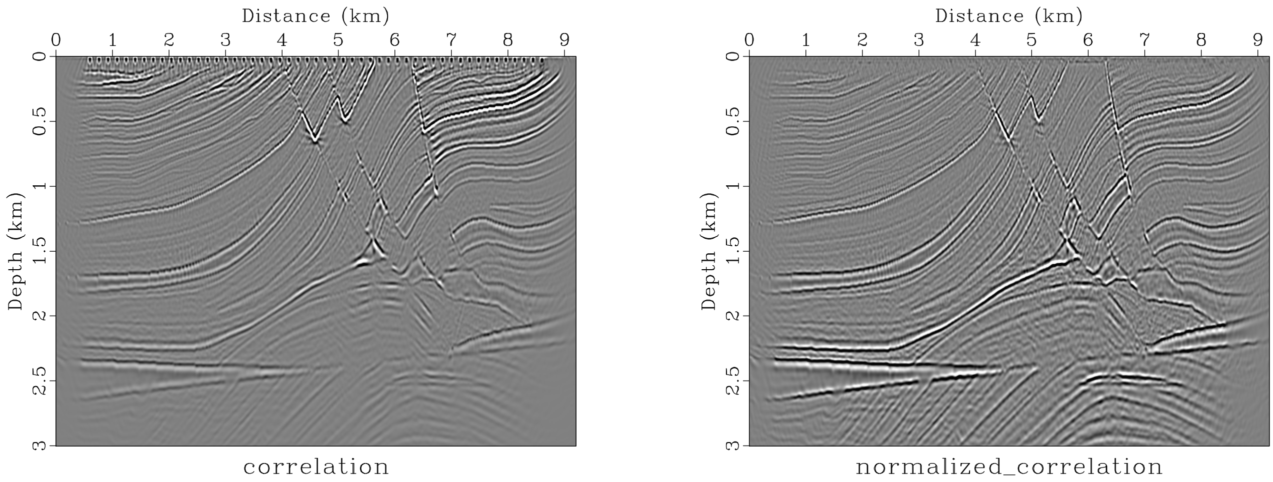


Figure 3: correlation imaging condition with and without normalization

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

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

in which $\sigma^2$ 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]

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

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

$$\mathbf{S} = \mathbf{v}p = \nabla p \frac{\mathrm{d}p}{\mathrm{d}t} p = (v_x p, v_z p). \tag{58}$$

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:

$$\gamma = \arccos \frac{\mathbf{S}_s \cdot \mathbf{S}_r}{|\mathbf{S}_s||\mathbf{S}_r|} \tag{59}$$

The incident angle (or reflective angle) is half of $\gamma$, namely,

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

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

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

where

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

Costa et al. (2009) modified the weight as

$$W(\theta) = cos^3(\theta/2). \tag{63}$$

# 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

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

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., $\mathsf{x}_r \subset \mathsf{x}$, where $\mathsf{x}_r = [(\mathbf{x}_r)_r]$, $\mathsf{x} = [(\mathbf{x}_i)_i], i = 1, 2, \ldots, M$.

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

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

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

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

Briefly speaking, it is

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

Thus,

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

where

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

and

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

$\nabla E_{\mathbf{m}}$ 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),

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

That is to say,

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

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 $\mathbf{H}$:

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

In matrix form

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

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., $\mathbf{H}_a = \mathtt{Re}[\mathbf{J}^\dagger\mathbf{J}]$, is referred to as the approximate Hessian. It is the auto-correlation of the derivative wavefield. Eq. (68) becomes

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

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

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

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:

$$\Delta\mathbf{m} = -\mu^{-1}\nabla E_{\mathbf{m}} = -\alpha\nabla E_{\mathbf{m}} = -\alpha\mathrm{Re}\left[\mathbf{J}^{\dagger}\Delta\mathbf{p}\right]. \tag{77}$$

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

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

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

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

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

## 4.2 Conjugate gradient (CG) implementation

The gradient-like method can be summarized as

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

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

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

There are many ways to compute $\beta_k$:

$$\begin{cases} \beta_k^{HS} = \dfrac{\langle\nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1})\rangle}{\langle\mathbf{d}_{k-1}, \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1})\rangle} \\ \beta_k^{FR} = \dfrac{\langle\nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k)\rangle}{\langle\nabla E(\mathbf{m}_{k-1}), \nabla E(\mathbf{m}_{k-1})\rangle} \\ \beta_k^{PRP} = \dfrac{\langle\nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1})\rangle}{\langle\nabla E(\mathbf{m}_{k-1}), \nabla E(\mathbf{m}_{k-1})\rangle} \\ \beta_k^{CD} = -\dfrac{\langle\nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k)\rangle}{\langle\mathbf{d}_{k-1}, \nabla E(\mathbf{m}_{k-1})\rangle} \\ \beta_k^{DY} = \dfrac{\langle\nabla E(\mathbf{m}_k), \nabla E(\mathbf{m}_k)\rangle}{\langle\mathbf{d}_{k-1}, \nabla E(\mathbf{m}_k) - \nabla E(\mathbf{m}_{k-1})\rangle} \end{cases} \tag{82}$$

In practice, we suggest to use

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

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

• Approach 1: Currently, the objective function is

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

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

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

• Approach 2: Recall that

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

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 $\epsilon$. In practice, we chose an $\epsilon$ such that

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

### 4.3 CG method for iteratively reweighted least squares (IRLS)

In Eq. (64), define $\mathbf{r} = \Delta \mathbf{p} = \mathbf{F}\mathbf{m} - \mathbf{p}_{obs}$ in which we denote forward modelling $\mathbf{f}(\mathbf{m}) = \mathbf{F}\mathbf{m}$ for brevity. It leads to minimizing the $\ell_2$-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

```
initialize m, β ← 0, g_1 ← 0
for iter=0...itermax-1 do
    g_0 ← g_1
    r ← Fm − p_obs
    compute objective function, if converged, break
    g_1 ← F†r
    compute β according to Eq. (82)
    cg ← −g_1 + βcg
    compute α according to Eq. (85) or (88)
    m ← m + αcg
end for
```

As a matter of fact, $\ell_p$-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

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

with

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

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

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

followed by

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

with

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

It is easy to verify that

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

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

**Algorithm 2** CG method for IRLS solution

```
initialize m, β ← 0, g_1 ← 0
for iter=0...itermax1-1 do
    compute W_r according to Eq. (93)
    compute W_m according to Eq. (96)
    r ← W_r(F W_m m − p_obs)
    compute objective function, if converged, break
    g_0 ← g_1
    g_1 ← F^† r
    compute β according to Eq. (82)
    cg ← −g_1 + β cg
    compute α according to Eq. (85) or (88)
    m ← m + α cg
    m ← W_m m
end for
```

## 4.4 Fréchet derivative

Recall that the basic acoustic wave equation can be specified as

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

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

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

Thus the integral representation of the solution can be given by

$$\begin{aligned}p(\mathbf{x},t;\mathbf{x}_s)&=\int_V \mathrm{d}V(\mathbf{x}')\int \mathrm{d}t'\Gamma(\mathbf{x},t;\mathbf{x}_s,t')f_s(\mathbf{x}',t;\mathbf{x}_s)\\&=\int_V \mathrm{d}V(\mathbf{x}')\int \mathrm{d}t'\Gamma(\mathbf{x},t-t';\mathbf{x}_s,0)f_s(\mathbf{x}',t;\mathbf{x}_s)(Causility\ of\ Green\ function)\\&=\int_V \mathrm{d}V(\mathbf{x}')\Gamma(\mathbf{x},t;\mathbf{x}_s,0)*f_s(\mathbf{x}',t;\mathbf{x}_s)\end{aligned} \tag{100}$$

where $*$ denotes the convolution operator.

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

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

Note that

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

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

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

Using the Born approximation, Eq. (103) becomes

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

Again, based on integral representation, we obtain

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

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),

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

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

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

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

$$\int \mathrm{d}t[g(t)*f(t)]h(t)=\int \mathrm{d}t f(t)[g(-t)*h(t)].$$

Then, Eq. (71) becomes

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

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,

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

# A Forward modelling with PA: CUDA code

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

```
    int id=i1+i2*nz;

    __shared__ float s_p0[Block_Size1+2][Block_Size2+2];
    __shared__ float s_p1[Block_Size1+2][Block_Size2+2];
    s_p0[threadIdx.x+1][threadIdx.y+1]=p0[id];
    s_p1[threadIdx.x+1][threadIdx.y+1]=p1[id];
    if(threadIdx.x==0)
    {
        if(blockIdx.x>0)     { s_p0[threadIdx.x][threadIdx.y+1]=p0[id
            -1];    s_p1[threadIdx.x][threadIdx.y+1]=p1[id-1];}
        else                 { s_p0[threadIdx.x][threadIdx.y+1]=0.0f;
                    s_p1[threadIdx.x][threadIdx.y+1]=0.0f;}
    }
    if(threadIdx.x==blockDim.x-1)
    {
        if(blockIdx.x<gridDim.x-1)     { s_p0[threadIdx.x+2][threadIdx.
            y+1]=p0[id+1];   s_p1[threadIdx.x+2][threadIdx.y+1]=p1[id+1];}
        else                           { s_p0[threadIdx.x+2][threadIdx.
            y+1]=0.0f;       s_p1[threadIdx.x+2][threadIdx.y+1]=0.0f;}
    }
    if(threadIdx.y==0)
    {
        if(blockIdx.y>0)     { s_p0[threadIdx.x+1][threadIdx.y]=p1[id
            -nz];    s_p1[threadIdx.x+1][threadIdx.y]=p1[id-nz];}
        else                 { s_p0[threadIdx.x+1][threadIdx.y]=0.0f;
                    s_p1[threadIdx.x+1][threadIdx.y]=0.0f;}
    }
    if(threadIdx.y==blockDim.y-1)
    {
        if(blockIdx.y<gridDim.y-1)     { s_p0[threadIdx.x+1][threadIdx.
            y+2]=p1[id+nz];  s_p1[threadIdx.x+1][threadIdx.y+2]=p1[id+nz
            ];}
        else                           { s_p0[threadIdx.x+1][threadIdx.
            y+2]=0.0f;       s_p1[threadIdx.x+1][threadIdx.y+2]=0.0f;}
    }
    __syncthreads();

    float c1=vel[id]*dtx2*(s_p1[threadIdx.x+2][threadIdx.y+1]-2.0*s_p1[
        threadIdx.x+1][threadIdx.y+1]+s_p1[threadIdx.x][threadIdx.y+1]);
    float c2=vel[id]*dtz2*(s_p1[threadIdx.x+1][threadIdx.y+2]-2.0*s_p1[
        threadIdx.x+1][threadIdx.y+1]+s_p1[threadIdx.x+1][threadIdx.y]);

    if(i1==0)// left boundary
    {
        c1=sqrtf(vel[id]*dtx2)*(-s_p1[threadIdx.x+1][threadIdx.y+1]+s_p1
            [threadIdx.x+2][threadIdx.y+1]
                    +s_p0[threadIdx.x+1][threadIdx.y+1]-s_p0
                        [threadIdx.x+2][threadIdx.y+1]);
        if(i2>0 && i2<nx-1) c2=0.5*c2;
    }
    if(i1==nz-1) // right boundary
    {
        c1=sqrtf(vel[id]*dtx2)*(s_p1[threadIdx.x][threadIdx.y+1]-s_p1[
            threadIdx.x+1][threadIdx.y+1]
                    -s_p0[threadIdx.x][threadIdx.y+1]+s_p0[
                        threadIdx.x+1][threadIdx.y+1]);
        if(i2>0 && i2<nx-1) c2=0.5*c2;
    }
    if(i2==0)// top boundary
    {
        if(i1>0 && i1<nz-1) c1=0.5*c1;
```

```
                c2=sqrtf(vel[id]*dtz2)*(-s_p1[threadIdx.x+1][threadIdx.y+1]+s_p1
                    [threadIdx.x+1][threadIdx.y+2]
                                            +s_p0[threadIdx.x+1][threadIdx.y+1]-s_p0
                                                [threadIdx.x+1][threadIdx.y+2]);

        }
        if(i2==nx-1) // bottom boundary
        {
                if(i1>0 && i1<nz-1) c1=0.5*c1;
                c2=sqrtf(vel[id]*dtz2)*(s_p1[threadIdx.x+1][threadIdx.y]-s_p1[
                    threadIdx.x+1][threadIdx.y+1]
                                            -s_p0[threadIdx.x+1][threadIdx.y]+s_p0[
                                                threadIdx.x+1][threadIdx.y+1]);
        }

        p0[id]=2*s_p1[threadIdx.x+1][threadIdx.y+1]-s_p0[threadIdx.x+1][
            threadIdx.y+1]+c1+c2;
}
```

## B Compute 2N-order difference coefficients: MATLAB code

```
function c=FindCoeff(NJ)
% Input NJ means that you are using 2*N-order difference strategy
% Example:
%       format long
%       NJ=10;
%       c=FindCoeff(NJ)
N=NJ/2;
x=zeros(N,1);
b=zeros(N,1);    b(1)=1;
c=b;
for k=1:N
    x(k)=(2*k-1)^2;
end

for k=1:N-1
    for i=N:-1:k+1
        b(i)=b(i)-x(k)*b(i-1);
    end
end

for k=N-1:-1:1
    for i=k+1:N
        b(i)=b(i)/(x(i)-x(i-k));
    end
    for i=k:N-1
        b(i)=b(i)-b(i+1);
    end
end
for k=1:N
    c(k)=b(k)/(2*k-1);
end
```

## C RTM: CUDA code

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

```

1) Coordinate configuration of seismic data:
                    y (3rd dim: *.z)
                   /
                  /
                 /
                /
               /
              o-------------------> x (2nd dim: *.y)
              |
              |
              |
              |
              |
              z (1st dim: *.x)
        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 RTM 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
```

```
#ifndef MAX
#define MAX(x,y) ((x) > (y) ? (x) : (y))
#endif
#ifndef MIN
#define MIN(x,y) ((x) < (y) ? (x) : (y))
#endif
#ifndef true
#define true    (1)
#endif
#ifndef false
#define false   (0)
#endif
#ifndef EPS
#define EPS     1.0e-15f
#endif

#define PI      3.141592653589793f
#define Block_Size1 16          // 1st dim block size
#define Block_Size2 16          // 2nd dim block size
const int npml=32;
const int NJ=6;                 // finite difference order NJ=2*k
const int nbell=3;              // radius of Gaussian bell
const bool csdgather=true;      // common shot gather (CSD) or not

#include "cuda_kernels.cu"
/*
const bool csdgather=false;     // common shot gather (CSD) or not
char *model_file="sm_segsalt210x675.bin";
const int       nz1=210;
const int       nx1=675;
const float     dx=20.0;
const float     dz=20.0;
const float     fm=30.0;
const float     dt=0.002;
const int       nt=3200;
const int       ns=30;
const int       ng=675;

int jsx=20;     //source x-axis jump interval
int jsz=0;      //source z-axis jump interval
int jgx=1;      //geophone x-axis jump interval
int jgz=0;      //geophone z-axis jump interval
int sxbeg=35;   //x-begining index of source, starting from 0
int szbeg=1;    //z-begining index of source, starting from 0
int gxbeg=0;    //x-begining index of geophone, starting from 0
int gzbeg=2;    //z-begining index of geophone, starting from 0
*/
/*
const bool csdgather=false;     // common shot gather (CSD) or not
char *model_file="marm240x737.bin";
const int       nz1=240;
const int       nx1=737;
const float     dx=12.5;
const float     dz=12.5;
const float     fm=20.0;
const float     dt=0.001;
const int       nt=3700;
const int       ns=21;
const int       ng=737;

int jsx=32;
```

```
int jsz=0;
int jgx=1;
int jgz=0;
int sxbeg=45;//x-begin point of source, index starting from 0
int szbeg=1;//z-begin point of source, index starting from 0
int gxbeg=0;//x-begin point of geophone, index starting from 0
int gzbeg=2;//z-begin point of geophone, index starting from 0

*/
/*
char *model_file="syn320x320.bin";
const int       nz1=320;
const int       nx1=320;
const float     dx=5.0;
const float     dz=5.0;
const float     fm=25.0;
const float     dt=0.001;
const int       nt=1800;
const int       ns=11;
const int       ng=60;

int jsx=26;     //source x-axis jump interval
int jsz=0;      //source z-axis jump interval
int jgx=1;      //geophone x-axis jump interval
int jgz=0;      //geophone z-axis jump interval
int sxbeg=30;   //x-begining index of source, starting from 0
int szbeg=1;    //z-begining index of source, starting from 0
int gxbeg=0;    //x-begining index of geophone, starting from 0
int gzbeg=2;    //z-begining index of geophone, starting from 0
*/


char *model_file="marm751x2301.bin";
const int       nz1=751;
const int       nx1=2301;
const float     dx=4.0;
const float     dz=4.0;
const float     fm=20.0;
const float     dt=0.0003;
const int       nt=13000;       // 10000
const int       ns=101;         // 41
const int       ng=301;

int jsx=20;     //source x-axis jump interval
int jsz=0;      //source z-axis jump interval
int jgx=1;      //geophone x-axis jump interval
int jgz=0;      //geophone z-axis jump interval
int sxbeg=150;  //x-begining index of source, starting from 0
int szbeg=2;    //z-begining index of source, starting from 0
int gxbeg=0;    //x-begining index of geophone, starting from 0
int gzbeg=3;    //z-begining index of geophone, starting from 0


const float     vmute=1500;
const float     _dx=1.0/dx;
const float     _dz=1.0/dz;
const int       nt_h=0.65*nt; // 65% points allocated on host using zero-copy pinned memory, the rest on device

static int      nz, nx, nnz, nnx, N;
static dim3     dimbbell, dimg0, dimb0;
```

```
static dim3     dimglr1, dimblr1, dimglr2, dimblr2;//lr=left and right
static dim3     dimgtb1, dimbtb1, dimgtb2, dimbtb2;//tb=top and bottom


// variables on host
float   *seis, *v0, *vel, *p;
// variables on device
int     *d_Sxz, *d_Gxz;                         // set source and geophone position
float   *d_bell,*d_wlt, *d_dobs,  *d_vel;       // bell, wavelet, seismograms, velocity (vel)
float   *d_usp,*d_sp0, *d_sp1, *d_svx, *d_svz;  // p, vx, vz for sources
float   *d_ugp,*d_gp0, *d_gp1, *d_gvx, *d_gvz;  // p, vx, vz for geophones
float   *d_bx1, *d_bx2, *d_bz1, *d_bz2;         // PML ABC coefficients for p and v (vx, vz)
float   *d_convpx, *d_convpz, *d_convvx, *d_convvz;// auxiliary variables to decay p and v in PML zone
float   *d_Iss, *d_Isg, *d_I1,*d_I2;            // I1: image without normalization; I2: normalized image;
float   *h_boundary, *d_boundary;               // boundary on host and device
float   *ptr=NULL;

void matrix_transpose(float *matrix, int nx, int nz)
{
        float *tmp=(float*)malloc(nx*nz*sizeof(float));
        if (tmp==NULL) {printf("out of memory!"); exit(1);}
        for(int iz=0; iz<nz; iz++){
                for(int ix=0; ix<nx; ix++){
                        tmp[iz+nz*ix]=matrix[ix+nx*iz];
                }
        }
        memcpy(matrix, tmp, nx*nz*sizeof(float));
        free(tmp);
}


// a: size=nz1*nx1; b: size=nnz*nnx;
void expand(float *a, float *b, int npml, int nnz, int nnx, int nz1, int nx1)
{/*< expand domain of 'a' to 'b' >*/
    int iz,ix;
    for     (ix=0;ix<nx1;ix++) {
        for  (iz=0;iz<nz1;iz++) {
            b[(npml+ix)*nnz+(npml+iz)] = a[ix*nz1+iz];
        }
    }
    for     (ix=0; ix<nnx; ix++) {
        for  (iz=0; iz<npml; iz++)       b[ix*nnz+iz] = b[ix*nnz+npml];//top
        for  (iz=nz1+npml; iz<nnz; iz++) b[ix*nnz+iz] = b[ix*nnz+npml+nz1-1];//bottom
    }

    for (iz=0; iz<nnz; iz++){
        for(ix=0; ix<npml; ix++)         b[ix*nnz+iz] = b[npml*nnz+iz];//left
        for(ix=npml+nx1; ix<nnx; ix++)   b[ix*nnz+iz] = b[(npml+nx1-1)*nnz+iz];//right
    }
}

// from: size=nnz*nnx; to: size=nz1*nx1;
void window(float *from, float *to, int npml, int nnz, int nnx, int nz1, int nx1)
```

```
{
        int ix, iz;
        for(ix=0;ix<nx1;ix++){
                for(iz=0;iz<nz1;iz++){
                        to[iz+ix*nz1]=from[(iz+npml)+(ix+npml)*nnz];
                }
        }
}


void check_gird_sanity()
{
        float C;
        if(NJ==2) C=1;
        else if (NJ==4)         C=0.857;
        else if (NJ==6)         C=0.8;
        else if (NJ==8)         C=0.777;
        else if (NJ==10)        C=0.759;

        float maxvel=vel[0], minvel=vel[0];
        for(int i=0; i<N; i++)  {
                if(vel[i]>maxvel) maxvel=vel[i];
                if(vel[i]<minvel) minvel=vel[i];
        }
        float tmp=dt*maxvel*sqrt(1.0/(dx*dx)+1.0/(dz*dz));

        if (tmp>=C) printf("Stability condition not satisfied!\n");
        if (fm>=minvel/(5*max(dx,dz))) printf("Dispersion relation not satisfied
            !\n");
}


void device_alloc()
{
        cudaMalloc(&d_bell,     (2*nbell+1)*(2*nbell+1)*sizeof(float));
        cudaMalloc(&d_Sxz,      ns*sizeof(int));
        cudaMalloc(&d_Gxz,      ng*sizeof(int));
        cudaMalloc(&d_wlt,      nt*sizeof(float));
        cudaMalloc(&d_dobs,     ng*nt*sizeof(float));
        cudaMalloc(&d_vel,      N*sizeof(float));
        cudaMalloc(&d_usp,      N*sizeof(float));
        cudaMalloc(&d_sp0,      N*sizeof(float));
        cudaMalloc(&d_sp1,      N*sizeof(float));
        cudaMalloc(&d_svx,      N*sizeof(float));
        cudaMalloc(&d_svz,      N*sizeof(float));
        cudaMalloc(&d_ugp,      N*sizeof(float));
        cudaMalloc(&d_gp0,      N*sizeof(float));
        cudaMalloc(&d_gp1,      N*sizeof(float));
        cudaMalloc(&d_gvx,      N*sizeof(float));
        cudaMalloc(&d_gvz,      N*sizeof(float));
        cudaMalloc(&d_bx1,      2*npml*nnz*sizeof(float));// left and right ABC
            coefficients for p
        cudaMalloc(&d_bz1,      2*npml*nnx*sizeof(float));// top and bottom ABC
            coefficients for p
        cudaMalloc(&d_bx2,      2*npml*nnz*sizeof(float));// left and right ABC
            coefficients for v (vx, vz)
        cudaMalloc(&d_bz2,      2*npml*nnx*sizeof(float));// top and bottom ABC
            coefficients for v (vx, vz)
        cudaMalloc(&d_convpx,   2*npml*nnz*sizeof(float));//  (left and right)
        cudaMalloc(&d_convpz,   2*npml*nnx*sizeof(float));//  (top and bottom)
        cudaMalloc(&d_convvx,   2*npml*nnz*sizeof(float));//  (left and right)
```

```
        cudaMalloc(&d_convvz,   2*npml*nnx*sizeof(float));//  (top and bottom)
        cudaMalloc(&d_Iss,      N*sizeof(float));
        cudaMalloc(&d_Isg,      N*sizeof(float));
        cudaMalloc(&d_I1,       N*sizeof(float));
        cudaMalloc(&d_I2,       N*sizeof(float));
        cudaHostAlloc(&h_boundary, nt_h*2*(NJ-1)*(nnx+nnz)*sizeof(float),
           cudaHostAllocMapped);
        cudaMalloc(&d_boundary, (nt-nt_h)*2*(NJ-1)*(nnx+nnz)*sizeof(float));

        cudaError_t err = cudaGetLastError ();
        if (cudaSuccess != err)
        printf("Cuda error: Failed to allocate required memory!: %s",
           cudaGetErrorString(err));
}


void device_free()
{
        cudaFree(d_bell);
        cudaFree(d_Sxz);
        cudaFree(d_Gxz);
        cudaFree(d_wlt);
        cudaFree(d_dobs);
        cudaFree(d_vel);
        cudaFree(d_usp);
        cudaFree(d_sp0);
        cudaFree(d_sp1);
        cudaFree(d_svx);
        cudaFree(d_svz);
        cudaFree(d_ugp);
        cudaFree(d_gp0);
        cudaFree(d_gp1);
        cudaFree(d_gvx);
        cudaFree(d_gvz);
        cudaFree(d_bx1);
        cudaFree(d_bx2);
        cudaFree(d_bz1);
        cudaFree(d_bz2);
        cudaFree(d_convpx);
        cudaFree(d_convpz);
        cudaFree(d_convvx);
        cudaFree(d_convvz);
        cudaFree(d_Iss);
        cudaFree(d_Isg);
        cudaFree(d_I1);
        cudaFree(d_I2);
        cudaFreeHost(h_boundary);
        cudaFree(d_boundary);

        cudaError_t err = cudaGetLastError ();
        if (cudaSuccess != err)
        printf("Cuda error: Failed to free the allocated memory!: %s",
           cudaGetErrorString(err));
}

//up:laplace(p)
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)
{
        cudaMemset(d_up,        0,      N*sizeof(float));
        cudaMemset(d_p0,        0,      N*sizeof(float));
```

```
    cudaMemset(d_p1,        0,      N*sizeof(float));
    cudaMemset(d_vx,        0,      N*sizeof(float));
    cudaMemset(d_vz,        0,      N*sizeof(float));
    cudaMemset(d_convpx,    0,      2*npml*nnz*sizeof(float));
    cudaMemset(d_convpz,    0,      2*npml*nnx*sizeof(float));
    cudaMemset(d_convvx,    0,      2*npml*nnz*sizeof(float));
    cudaMemset(d_convvz,    0,      2*npml*nnx*sizeof(float));

    cudaError_t err = cudaGetLastError ();
    if (cudaSuccess != err)
    printf("Cuda error: Failed to initialize the wavefield variables!: %s",
        cudaGetErrorString(err));
}

void forward_laplacian(float *d_up, float *d_p1, float *d_vx, float *d_vz, float
      *d_convvx, float *d_convvz, float *d_convpx, float *d_convpz, float *d_bx1,
    float *d_bz1, float *d_bx2, float *d_bz2)
{
    // p0: p{it-1};
    // p1: p{it}; up: laplacian of p1
    // p{it+1}-->p0
    if (NJ==2)      {
        cuda_forward_v_2<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz,
            npml, nnz, nnx);
        cuda_PML_vz_2<<<dimgtb1, dimbtb1>>>(d_p1, d_convpz, d_bz2, d_vz,
             _dz, npml, nnz, nnx);
        cuda_PML_vx_2<<<dimglr1, dimblr1>>>(d_p1, d_convpx, d_bx2, d_vx,
             _dx, npml, nnz, nnx);
        cuda_forward_up_2<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz,
            npml, nnz, nnx);
        cuda_PML_upz_2<<<dimgtb1, dimbtb1>>>(d_up, d_convvz, d_bz1, d_vz
            , _dz, npml, nnz, nnx);
        cuda_PML_upx_2<<<dimglr1, dimblr1>>>(d_up, d_convvx, d_bx1, d_vx
            , _dx, npml, nnz, nnx);
    }else if (NJ==4){
        cuda_forward_v_4<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz,
            npml, nnz, nnx);
        cuda_PML_vz_4<<<dimgtb1, dimbtb1>>>(d_p1, d_convpz, d_bz2, d_vz,
             _dz, npml, nnz, nnx);
        cuda_PML_vx_4<<<dimglr1, dimblr1>>>(d_p1, d_convpx, d_bx2, d_vx,
             _dx, npml, nnz, nnx);
        cuda_forward_up_4<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz,
            npml, nnz, nnx);
        cuda_PML_upz_4<<<dimgtb1, dimbtb1>>>(d_up, d_convvz, d_bz1, d_vz
            , _dz, npml, nnz, nnx);
        cuda_PML_upx_4<<<dimglr1, dimblr1>>>(d_up, d_convvx, d_bx1, d_vx
            , _dx, npml, nnz, nnx);
    }else if (NJ==6){
        cuda_forward_v_6<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz,
            npml, nnz, nnx);
        cuda_PML_vz_6<<<dimgtb1, dimbtb1>>>(d_p1, d_convpz, d_bz2, d_vz,
             _dz, npml, nnz, nnx);
        cuda_PML_vx_6<<<dimglr1, dimblr1>>>(d_p1, d_convpx, d_bx2, d_vx,
             _dx, npml, nnz, nnx);
        cuda_forward_up_6<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz,
            npml, nnz, nnx);
        cuda_PML_upz_6<<<dimgtb1, dimbtb1>>>(d_up, d_convvz, d_bz1, d_vz
            , _dz, npml, nnz, nnx);
        cuda_PML_upx_6<<<dimglr1, dimblr1>>>(d_up, d_convvx, d_bx1, d_vx
            , _dx, npml, nnz, nnx);
    }else if (NJ==8){
```

```
		cuda_forward_v_8<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_PML_vz_8<<<dimgtb1, dimbtb1>>>(d_p1, d_convpz, d_bz2, d_vz, _dz, npml, nnz, nnx);
		cuda_PML_vx_8<<<dimglr1, dimblr1>>>(d_p1, d_convpx, d_bx2, d_vx, _dx, npml, nnz, nnx);
		cuda_forward_up_8<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_PML_upz_8<<<dimgtb1, dimbtb1>>>(d_up, d_convvz, d_bz1, d_vz, _dz, npml, nnz, nnx);
		cuda_PML_upx_8<<<dimglr1, dimblr1>>>(d_up, d_convvx, d_bx1, d_vx, _dx, npml, nnz, nnx);
	}else if (NJ==10){
		cuda_forward_v_10<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_PML_vz_10<<<dimgtb1, dimbtb1>>>(d_p1, d_convpz, d_bz2, d_vz, _dz, npml, nnz, nnx);
		cuda_PML_vx_10<<<dimglr1, dimblr1>>>(d_p1, d_convpx, d_bx2, d_vx, _dx, npml, nnz, nnx);
		cuda_forward_up_10<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_PML_upz_10<<<dimgtb1, dimbtb1>>>(d_up, d_convvz, d_bz1, d_vz, _dz, npml, nnz, nnx);
		cuda_PML_upx_10<<<dimglr1, dimblr1>>>(d_up, d_convvx, d_bx1, d_vx, _dx, npml, nnz, nnx);
	}
}

void backward_laplacian(float *d_up, float *d_p1, float *d_vx, float *d_vz)
{
	// p0: p{it-1};
	// p1: p{it}; up: laplacian of p1
	// p{it+1}-->p0
	if (NJ==2){
		cuda_forward_v_2<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_forward_up_2<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
	}
	else if (NJ==4) {
		cuda_forward_v_4<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_forward_up_4<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
	}
	else if (NJ==6) {
		cuda_forward_v_6<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_forward_up_6<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
	}
	else if (NJ==8) {
		cuda_forward_v_8<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
		cuda_forward_up_8<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
	}
	else if (NJ==10){
		cuda_forward_v_10<<<dimg0, dimb0>>>(d_p1, d_vx, d_vz, _dx, _dz, npml, nnz, nnx);
```

```
            cuda_forward_up_10<<<dimg0, dimb0>>>(d_up, d_vx, d_vz, _dx, _dz,
                npml, nnz, nnx);
        }
}


int main(int argc, char *argv[])
{
        nx=(int)((nx1+Block_Size1-1)/Block_Size1)*Block_Size1;
        nz=(int)((nz1+Block_Size2-1)/Block_Size2)*Block_Size2;
        nnz = 2*npml+nz;
        nnx = 2*npml+nx;
        N=nnz*nnx;
        dimbbell=dim3(2*nbell+1,2*nbell+1);
        dimb0=dim3(Block_Size1, Block_Size2);   dimg0=dim3(nnz/Block_Size1, nnx/
            Block_Size2);
        dimblr1=dim3(Block_Size1, 32);          dimglr1=dim3(nnz/Block_Size1, 2)
            ;
        dimbtb1=dim3(32, Block_Size2);          dimgtb1=dim3(2, nnx/Block_Size2)
            ;
        dimblr2=dim3(nnz/Block_Size1,(NJ+15)/16); dimglr2=dim3(Block_Size1, 16);
        dimbtb2=dim3(16, Block_Size2);          dimgtb2=dim3((NJ+15)/16, nnx/
            Block_Size2);

        v0=(float*)malloc(nx1*nz1*sizeof(float));
        if (v0==NULL)   { printf("out of memory!"); exit(1);}
        seis=(float*)malloc(ng*nt*sizeof(float));
        if (seis==NULL) { printf("out of memory!"); exit(1);}
        vel=(float*)malloc(N*sizeof(float));
        if (vel==NULL)  { printf("out of memory!"); exit(1);}
        p=(float*)malloc(N*sizeof(float));
        if (p==NULL)    { printf("out of memory!"); exit(1);}
        memset(v0, 0, nz1*nx1*sizeof(float));
        memset(seis, 0, ng*nt*sizeof(float));
        memset(vel, 0, N*sizeof(float));
        memset(p, 0, N*sizeof(float));

        FILE *fp,*fp1;
        fp=fopen(model_file,"rb");
        if (fp==NULL) { printf("cannot open the model file"); exit(1);}
        fread(v0,sizeof(float),nz1*nx1,fp);
        fclose(fp);
        expand(v0, vel, npml, nnz, nnx, nz1, nx1);


        printf("NJ=%d \t\n",    NJ);
        printf("npml=%d \t\n",  npml);
        printf("nx1=%d   \t\n", nx1);
        printf("nz1=%d   \t\n", nz1);
        printf("nx=%d   \t\n",  nx);
        printf("nz=%d   \t\n",  nz);
        printf("nnx=%d  \t\n",  nnx);
        printf("nnz=%d  \t\n",  nnz);
        printf("dx=%g\t\t(m)\n", dx);
        printf("dz=%g\t\t(m)\n", dz);
        printf("dt=%g  \t(s)\n", dt);
        printf("fm=%g\t\t(Hz)\n", fm);
        printf("nt=%d\n",       nt);
        printf("ns=%d\n",       ns);
        printf("ng=%d\n",       ng);
        check_gird_sanity();
```

```
	cudaSetDevice(0);
	cudaError_t err = cudaGetLastError ();
	if (cudaSuccess != err)
	printf("Cuda error: Failed to initialize device: %s", cudaGetErrorString(err));
	device_alloc();

	float mstimer = 0;// timer unit: millionseconds
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	fp=fopen("img1.bin","wb");
	if (fp==NULL) { printf("cannot open the file\n"); exit(1);}
	fp1=fopen("img2.bin","wb");
	if (fp1==NULL) { printf("cannot open the file\n"); exit(1);}

	cuda_ini_bell<<<dim3(1,1),dim3(2*nbell+1,2*nbell+1)>>>(d_bell);
	cuda_ricker_wavelet<<<(nt+511)/512,512>>>(d_wlt, fm, dt, nt);
	if (!(sxbeg>=0 && szbeg>=0 && sxbeg+(ns-1)*jsx<nx && szbeg+(ns-1)*jsz<nz))
	{ printf("sources exceeds the computing zone!\n"); exit(1);}
	cuda_set_sg<<<(ns+255)/256, 256>>>(d_Sxz, sxbeg, szbeg, jsx, jsz, ns, npml, nnz);

	int distx=sxbeg-gxbeg;
	int distz=szbeg-gzbeg;
	if (csdgather) {
		//distance between source and geophone at the beginning
		if (!(gxbeg>=0 && gzbeg>=0 && gxbeg+(ng-1)*jgx<nx && gzbeg+(ng-1)*jgz<nz &&
		(sxbeg+(ns-1)*jsx)+(ng-1)*jgx-distx <nx  && (szbeg+(ns-1)*jsz)+(ng-1)*jgz-distz <nz))
		{ printf("geophones exceeds the computing zone!\n"); exit(1);}
	}
	else{
		if (!(gxbeg>=0 && gzbeg>=0 && gxbeg+(ng-1)*jgx<nx && gzbeg+(ng-1)*jgz<nz))
		{ printf("geophones exceeds the computing zone!\n"); exit(1);}
	}
	cuda_set_sg<<<(ng+255)/256, 256>>>(d_Gxz, gxbeg, gzbeg, jgx, jgz, ng, npml, nnz);

	//cuda_set2<<<dimg0, dimb0>>>(d_vel, 1700.0, 2000.0,nnz, nnx);
	cudaMemcpy(d_vel, vel, N*sizeof(float), cudaMemcpyHostToDevice);
	cudaMemset(d_Iss, 0,	N*sizeof(float));
	cudaMemset(d_Isg, 0,	N*sizeof(float));
	cudaMemset(d_I1, 0,	N*sizeof(float));
	cudaMemset(d_I2, 0,	N*sizeof(float));
	cuda_ini_abcz<<<dimgtb1, dimbtb1>>>(d_vel, d_bz1, d_bz2, dx, dz, dt, npml, nnz, nnx);
	cuda_ini_abcx<<<dimglr1, dimblr1>>>(d_vel, d_bx1, d_bx2, dx, dz, dt, npml, nnz, nnx);

	for(int is=0; is<ns; is++)
	{
		cudaEventRecord(start);

		cudaMemset(d_Isg,	0,	N*sizeof(float));
		cudaMemset(d_Iss,	0,	N*sizeof(float));
```

```
		cudaMemset(d_dobs,	0,	nt*ng*sizeof(float));
		cudaMemset(h_boundary,	0,	nt_h*2*(NJ-1)*(nnx+nnz)*sizeof(float));
		cudaMemset(d_boundary,	0,	(nt-nt_h)*2*(NJ-1)*(nnx+nnz)*sizeof(float));
		wavefield_ini(d_usp, d_sp0, d_sp1, d_svx, d_svz, d_convpx, d_convpz, d_convvx, d_convvz);
		if (csdgather)	{
			gxbeg=sxbeg+is*jsx-distx;
			cuda_set_sg<<<(ng+255)/256, 256>>>(d_Gxz, gxbeg, gzbeg, jgx, jgz, ng, npml, nnz);
		}
		for(int kt=0; kt<nt; kt++)
		{
			forward_laplacian(d_usp, d_sp1, d_svx, d_svz, d_convvx, d_convvz, d_convpx, d_convpz, d_bx1, d_bz1, d_bx2, d_bz2);
			cuda_step_forward<<<dimg0,dimb0>>>(d_vel, d_usp, d_sp0, d_sp1, dt, false, npml, nnz, nnx);
			cuda_add_bellwlt<<<dim3(1,1), dimbbell>>>(d_sp1, d_bell, &d_wlt[kt], &d_Sxz[is], 1, npml, nnz, nnx, true);
			ptr=d_sp0; d_sp0=d_sp1; d_sp1=ptr;


			cuda_record<<<(ng+255)/256, 256>>>(d_sp0, &d_dobs[kt*ng], d_Gxz, ng);
/*
			if (kt%50==0){
				cudaMemcpy(p, d_sp1, N*sizeof(float), cudaMemcpyDeviceToHost);
				fwrite(p, sizeof(float), N, fp);
			}
*/
			cuda_mute<<<(ng+511)/512, 512>>>(&d_dobs[kt*ng], gzbeg, szbeg, gxbeg, sxbeg+is*jsx, jgx, kt, 280, vmute, dt, dz, dx, ng);

			if(kt<nt_h) cudaHostGetDevicePointer(&ptr, &h_boundary[kt*2*(NJ-1)*(nnx+nnz)], 0);
			else  ptr=&d_boundary[(kt-nt_h)*2*(NJ-1)*(nnx+nnz)];
			cuda_rw_boundarytb<<<dimgtb2, dimbtb2>>>(ptr,   d_sp0, npml, nnz, nnx, NJ, false);
			cuda_rw_boundarylr<<<dimglr2, dimblr2>>>(&ptr[2*(NJ-1)*nnx], d_sp0, npml, nnz, nnx, NJ, false);
		}
/*
		cudaMemcpy(seis, d_dobs, nt*ng*sizeof(float), cudaMemcpyDeviceToHost);
		matrix_transpose(seis, ng, nt);// before: nx=ng; nz=nt; after: nx=nt; nz=ng;
		fwrite(seis, sizeof(float), nt*ng, fp);
*/

		ptr=d_sp0; d_sp0=d_sp1; d_sp1=ptr;
		wavefield_ini(d_ugp,d_gp0, d_gp1, d_gvx, d_gvz, d_convpx, d_convpz, d_convvx, d_convvz);
		for(int kt=nt-1; kt>-1; kt--)
		{
			// read saved boundary
			if(kt<nt_h) cudaHostGetDevicePointer(&ptr, &h_boundary[kt*2*(NJ-1)*(nnx+nnz)], 0);
```

```
				else	ptr=&d_boundary[(kt-nt_h)*2*(NJ-1)*(nnx+nnz)];
				cuda_rw_boundarytb<<<dimgtb2, dimbtb2>>>(ptr, d_sp1, npml, nnz, nnx, NJ, true);
				cuda_rw_boundarylr<<<dimglr2, dimblr2>>>(&ptr[2*(NJ-1)*nnx], d_sp1, npml, nnz, nnx, NJ, true);
				// subtract the wavelet
				cuda_add_bellwlt<<<dim3(1,1), dimbbell>>>(d_sp1, d_bell, &d_wlt[kt], &d_Sxz[is], 1, npml, nnz, nnx, false);
				backward_laplacian(d_usp, d_sp1, d_svx, d_svz);
				// backward time step source wavefield
				cuda_step_forward<<<dimg0,dimb0>>>(d_vel, d_usp, d_sp0, d_sp1, dt, false, npml, nnz, nnx);
				ptr=d_sp0; d_sp0=d_sp1; d_sp1=ptr;

				// backward time step receiver wavefield
				forward_laplacian(d_ugp, d_gp1, d_gvx, d_gvz, d_convvx, d_convvz, d_convpx, d_convpz, d_bx1, d_bz1, d_bx2, d_bz2);
				cuda_step_forward<<<dimg0,dimb0>>>(d_vel, d_ugp, d_gp0, d_gp1, dt, false, npml, nnz, nnx);
				// add receiver term
				cuda_add_source<<<(ng+255)/256,256>>>(d_gp0, &d_dobs[kt*ng], d_Gxz, ng, true);
				ptr=d_gp0; d_gp0=d_gp1; d_gp1=ptr;

				cuda_cross_correlate<<<dimg0, dimb0>>>(d_Isg, d_Iss, d_sp0, d_gp0, npml, nnz, nnx);
			}
			cuda_imaging<<<dimg0, dimb0>>>(d_Isg, d_Iss, d_I1, d_I2, npml, nnz, nnx);

			cudaEventRecord(stop);
			cudaEventSynchronize(stop);
			cudaEventElapsedTime(&mstimer, start, stop);
			printf("%d shot finished: %f (s)\n",is+1, mstimer*1e-3);
		}
		cuda_laplace_filter<<<dimg0,dimb0>>>(d_I1,d_usp,_dz,_dx, npml, nnz, nnx);
		cuda_laplace_filter<<<dimg0,dimb0>>>(d_I2,d_ugp,_dz,_dx, npml, nnz, nnx);
		cudaMemcpy(p, d_usp, N*sizeof(float), cudaMemcpyDeviceToHost);
		window(p, v0, npml, nnz, nnx, nz1, nx1);
		fwrite(v0, sizeof(float), nz1*nx1, fp);
		fclose(fp);
		cudaMemcpy(p, d_ugp, N*sizeof(float), cudaMemcpyDeviceToHost);
		window(p, v0, npml, nnz, nnx, nz1, nx1);
		fwrite(v0, sizeof(float), nz1*nx1, fp1);
		fclose(fp1);

		cudaEventDestroy(start);
		cudaEventDestroy(stop);

		free(seis);
		free(v0);
		free(vel);
		free(p);
		device_free();

		return 0;
}
```

```
/* cuda_kernels.cu
  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.
*/
__global__ void cuda_set(float*p, float c, int nnz, int nnx)
{
        int i1=threadIdx.x+blockDim.x*blockIdx.x;
        int i2=threadIdx.y+blockDim.y*blockIdx.y;
        int id=i1+i2*nnz;

        if (i1>=0 && i1<nnz && i2>=0 && i2<nnx) p[id]=c;
}

// set part of p[nnz][nnx]=c1, part of p[nnz][nnx]=c2;
__global__ void cuda_set2(float*p, float c1, float c2, int nnz, int nnx)
{
        int i1=threadIdx.x+blockDim.x*blockIdx.x;
        int i2=threadIdx.y+blockDim.y*blockIdx.y;
        int id=i1+i2*nnz;

        if (i1<nnz/2 && i2<nnx) p[id]=c1;
        else p[id]=c2;
}

// set the positions of sources and geophones in whole domain
__global__ void cuda_set_sg(int *sxz, int sxbeg, int szbeg, int jsx, int jsz,
    int ns, int npml, int nnz)
{
        int id=threadIdx.x+blockDim.x*blockIdx.x;
        if (id<ns) sxz[id]=nnz*(sxbeg+id*jsx+npml)+(szbeg+id*jsz+npml);
}

// generate ricker wavelet with time deley
__global__ void cuda_ricker_wavelet(float *wlt, float fm, float dt, int nt)
{
        int it=threadIdx.x+blockDim.x*blockIdx.x;
        float tmp = PI*fm*fabsf(it*dt-1.0/fm);  //delay the wavelet to exhibit
            all waveform
        tmp *=tmp;
        if (it<nt) wlt[it]= (1.0-2.0*tmp)*expf(-tmp);   // ricker wavelet at
            time: t=nt*dt
}

// add==true, add (inject) the source; add==false, subtract the source
__global__ void cuda_add_source(float *p, float *source, int *Sxz, int ns, bool
    add)
{
        int id=threadIdx.x+blockDim.x*blockIdx.x;
        if (id<ns)
        {
                if (add)        p[Sxz[id]]+=source[id];
                else            p[Sxz[id]]-=source[id];
        }
}


// record the seismogram at time kt
__global__ void cuda_record(float*p, float *seis_kt, int *Gxz, int ng)
{
```

```
	int id=threadIdx.x+blockDim.x*blockIdx.x;
	if (id<ng) seis_kt[id]=p[Gxz[id]];
}

// mute the direct arrival according to the given velocity vmute
__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)
{
	int id=threadIdx.x+blockDim.x*blockIdx.x;
	float a=dx*abs(gxbeg+id*jgx-sxc);
	float b=dz*(gzbeg-szbeg);
	float t0=sqrtf(a*a+b*b)/vmute;
	int ktt=int(t0/dt)+ntd;// ntd is manually added to obtain the best
	    muting effect.
	if (id<ng && kt<ktt) seis_kt[id]=0.0;
}

// initialize the PML coefficients along x direction
__global__ void cuda_ini_abcx(float *vel, float *bx1, float *bx2, float dx,
    float dz, float dt, int npml, int nnz, int nnx)
{
	// bx1: left and right PML ABC coefficients, decay p (px,pz) along x
	    direction
	// bx2: left and right PML ABC coefficients, decay v (vx,vz) along x
	    direction
	// only 2 blocks used horizontally, blockIdx.x=0, 1

	// id: position in top or bottom PML zone itself
	// blockIdx.x==0, left PML zone; blockIdx.x==1, right PML zone
	int i1=threadIdx.x+blockIdx.x*blockDim.x;
	int i2=blockIdx.y*npml+threadIdx.y;
	int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
	int ik=i1+nnz*i2;

	float Rc=1.0e-5f;
	float d=npml*MAX(dx,dz);
	float d0=-3.0f*vel[id]*logf(Rc)/d/2.0f;
	float tmp1, tmp2;

	if (i2<npml)	// left PML zone
	{
		tmp1=(float)(npml-i2);
		tmp2=tmp1-0.5f;

	}
	else		// right PML zone
	{
		tmp1=i2-npml+0.5f;
		tmp2=(tmp1+0.5f);
	}
	tmp1=tmp1/npml;
	tmp2=tmp2/npml;
	tmp1=tmp1*tmp1;
	tmp2=tmp2*tmp2;
	bx1[ik]=expf(-d0*tmp1*dt);
	bx2[ik]=expf(-d0*tmp2*dt);
}


// initialize the PML coefficients along z-axis
```

```
__global__ void cuda_ini_abcz(float *vel, float *bz1, float *bz2, float dx,
    float dz, float dt, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone

        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int id=nnz*i2+(blockIdx.x*(nnz-npml)+threadIdx.x);
        int ik=i1+2*npml*i2;

        float Rc=1.0e-5f;
        float d=npml*MAX(dx,dz);
        float d0=-3.0f*vel[id]*logf(Rc)/d/2.0f;
        float tmp1, tmp2;

        if (i1<npml)    // top PML zone
        {
                tmp1=(float)(npml-i1);
                tmp2=tmp1-0.5f;
        }
        else            // bottom PML zone
        {
                tmp1=i1-npml+0.5f;
                tmp2=(float)(tmp1+0.5f);
        }
        tmp1=tmp1/npml;
        tmp2=tmp2/npml;
        tmp1=tmp1*tmp1;
        tmp2=tmp2*tmp2;
        bz1[ik]=expf(-d0*tmp1*dt);
        bz2[ik]=expf(-d0*tmp2*dt);
}


__global__ void cuda_ini_bell(float *bell)
{
        int i1=threadIdx.x;
        int i2=threadIdx.y;
        int id=i1+i2*(2*nbell+1);
        float s = 0.5*nbell;
        bell[id]=expf(-((i1-nbell)*(i1-nbell)+(i2-nbell)*(i2-nbell))/s);
}

// inject Gaussian bell smoothed wavelet
// lauch configuration: <<<dim3(ns,1), dim3(2*nbell+1,2*nbell+1)>>>
// add==true, add (inject) the wavelet; add==false, subtract the wavelet
__global__ void cuda_add_bellwlt(float *p, float *bell, float *wlt, int *Sxz,
    int ns, int npml, int nnz, int nnx, bool add)
{
        int i1=threadIdx.x;
        int i2=threadIdx.y;
        int is=blockIdx.x;// source wavelet index

        if (is<ns)
```

```
        {
                if (add)          p[Sxz[is]+(i1-nbell)+(i2-nbell)*nnz]+=bell[i1+i2
                    *(2*nbell+1)]*wlt[is];
                else              p[Sxz[is]+(i1-nbell)+(i2-nbell)*nnz]-=bell[i1+i2
                    *(2*nbell+1)]*wlt[is];
        }
}

//================================================ NJ=2
    ======================================================
__global__ void cuda_forward_v_2(float *p, float *vx, float *vz, float _dx,
    float _dz, int npml, int nnz, int nnx)
{
        int i1=blockIdx.x*blockDim.x+threadIdx.x;
        int i2=blockIdx.y*blockDim.y+threadIdx.y;
        int id=i1+i2*nnz;

        __shared__ float s_p[Block_Size1+1][Block_Size2+1];
        s_p[threadIdx.x][threadIdx.y]=p[id];
        if (threadIdx.x>blockDim.x-2)
        {
                if (blockIdx.x<gridDim.x-1)     s_p[threadIdx.x+1][threadIdx.y]=
                    p[id+1];
                else                            s_p[threadIdx.x+1][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>blockDim.y-2)
        {
                if (blockIdx.y<gridDim.y-1)     s_p[threadIdx.x][threadIdx.y+1]=
                    p[id+nnz];
                else                            s_p[threadIdx.x][threadIdx.y
                    +1]=0.0f;
        }
        __syncthreads();

        float diff1=(s_p[threadIdx.x+1][threadIdx.y]-s_p[threadIdx.x][threadIdx.
            y]);// .x-->1st dim--> i1
        float diff2=(s_p[threadIdx.x][threadIdx.y+1]-s_p[threadIdx.x][threadIdx.
            y]);// .y-->2nd dim--> i2
        vz[id]=_dz*diff1;
        vx[id]=_dx*diff2;
}

__global__ void cuda_PML_vz_2(float *p, float *convpz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0,1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=(blockIdx.x*(nnz-npml)+threadIdx.x)+nnz*i2;

        __shared__ float s_p[33][Block_Size2];
```

```
        s_p[threadIdx.x][threadIdx.y]=p[id];
        if (threadIdx.x>30)
        {
                if (blockIdx.x<gridDim.x-1)     s_p[threadIdx.x+1][threadIdx.y]=p[id+1];
                else                            s_p[threadIdx.x+1][threadIdx.y]=0.0f;
        }
        __syncthreads();

        float diff1=(s_p[threadIdx.x+1][threadIdx.y]-s_p[threadIdx.x][threadIdx.y]);
        convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        vz[id]+=convpz[ik];
}

__global__ void cuda_PML_vx_2(float *p, float *convpx, float *bx, float *vx, float _dx, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+nnz*i2;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+   threadIdx.y);

        __shared__ float s_p[Block_Size1][33];
        s_p[threadIdx.x][threadIdx.y]=p[id];
        if (threadIdx.y>30)
        {
                if (blockIdx.y<gridDim.y-1)     s_p[threadIdx.x][threadIdx.y+1]=p[id+nnz];
                else                            s_p[threadIdx.x][threadIdx.y+1]=0.0f;
        }
        __syncthreads();

        float diff2=(s_p[threadIdx.x][threadIdx.y+1]-s_p[threadIdx.x][threadIdx.y]);
        convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
        vx[id]+=convpx[ik];
}

__global__ void cuda_forward_up_2(float *up, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int id=i1+i2*nnz;

        __shared__ float s_v1[Block_Size1+1][Block_Size2];
        __shared__ float s_v2[Block_Size1][Block_Size2+1];
        s_v1[threadIdx.x+1][threadIdx.y]=vz[id];
        s_v2[threadIdx.x][threadIdx.y+1]=vx[id];
```

```
        if (threadIdx.x<1)
        {
                if (blockIdx.x)                 s_v1[threadIdx.x][threadIdx.y]=
                    vz[id-1];
                else                            s_v1[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y<1)
        {
                if (blockIdx.y)                 s_v2[threadIdx.x][threadIdx.y]=
                    vx[id-nnz];
                else                            s_v2[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        __syncthreads();

        float diff1=(s_v1[threadIdx.x+1][threadIdx.y]-s_v1[threadIdx.x][
            threadIdx.y]);
        float diff2=(s_v2[threadIdx.x][threadIdx.y+1]-s_v2[threadIdx.x][
            threadIdx.y]);
        up[id]=_dz*diff1+_dx*diff2;
}



__global__ void cuda_PML_upz_2(float *up,  float *convvz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=(blockIdx.x*(nnz-npml)+  threadIdx.x)+   nnz*    i2;

        __shared__ float s_v1[33][Block_Size2];
        s_v1[threadIdx.x+1][threadIdx.y]=vz[id];
        if (threadIdx.x<1)
        {
                if (blockIdx.x)                 s_v1[threadIdx.x][threadIdx.y]=
                    vz[id-1];
                else                            s_v1[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        __syncthreads();

        float diff1=(s_v1[threadIdx.x+1][threadIdx.y]-s_v1[threadIdx.x][
            threadIdx.y]);
        convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        up[id]+=convvz[ik];
}

__global__ void cuda_PML_upx_2(float *up,  float *convvx, float *bx, float *vx,
    float _dx, int npml, int nnz, int nnx)
```

```
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+nnz*i2;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

        __shared__ float s_v2[Block_Size1][33];
        s_v2[threadIdx.x][threadIdx.y+1]=vx[id];
        if (threadIdx.y<1)
        {
                if (blockIdx.y)                 s_v2[threadIdx.x][threadIdx.y]=vx[id-nnz];
                else                            s_v2[threadIdx.x][threadIdx.y]=0.0f;
        }
        __syncthreads();

        float diff2=(s_v2[threadIdx.x][threadIdx.y+1]-s_v2[threadIdx.x][threadIdx.y]);
        convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
        up[id]+=convvx[ik];
}


//============================================= NJ=4 ==========================================================
__global__ void cuda_forward_v_4(float *p, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
        int i1=blockIdx.x*blockDim.x+threadIdx.x;
        int i2=blockIdx.y*blockDim.y+threadIdx.y;
        int id=i1+i2*nnz;

        __shared__ float s_p[Block_Size1+3][Block_Size2+3];
        s_p[threadIdx.x+1][threadIdx.y+1]=p[id];
        if (threadIdx.x<1)
        {
                if (blockIdx.x)                 s_p[threadIdx.x][threadIdx.y+1]=p[id-1];
                else                            s_p[threadIdx.x][threadIdx.y+1]=0.0f;
        }
        if (threadIdx.x>blockDim.x-3)
        {
                if (blockIdx.x<gridDim.x-1)     s_p[threadIdx.x+3][threadIdx.y+1]=p[id+2];
                else                            s_p[threadIdx.x+3][threadIdx.y+1]=0.0f;
        }
        if (threadIdx.y<1)
        {
```

```
                if (blockIdx.y)                 s_p[threadIdx.x+1][threadIdx.y]=
                    p[id-nnz];
                else                            s_p[threadIdx.x+1][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>blockDim.y-3)
        {
                if (blockIdx.y<gridDim.y-1)     s_p[threadIdx.x+1][threadIdx.y
                    +3]=p[id+2*nnz];
                else                            s_p[threadIdx.x+1][threadIdx.y
                    +3]=0.0f;
        }
        __syncthreads();

        float diff1=1.125f*(s_p[threadIdx.x+2][threadIdx.y+1]-s_p[threadIdx.x
            +1][threadIdx.y+1])
                        -0.041666666666667f*(s_p[threadIdx.x+3][threadIdx.y+1]-
                            s_p[threadIdx.x][threadIdx.y+1]);
        float diff2=1.125f*(s_p[threadIdx.x+1][threadIdx.y+2]-s_p[threadIdx.x
            +1][threadIdx.y+1])
                        -0.041666666666667f*(s_p[threadIdx.x+1][threadIdx.y+3]-
                            s_p[threadIdx.x+1][threadIdx.y]);
        vz[id]=_dz*diff1;
        vx[id]=_dx*diff2;
}

__global__ void cuda_PML_vz_4(float *p, float *convpz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

        __shared__ float s_p[35][Block_Size2];
        s_p[threadIdx.x+1][threadIdx.y]=p[id];
        if (threadIdx.x<1)
        {
                if (blockIdx.x)                 s_p[threadIdx.x][threadIdx.y]=p[
                    id-1];
                else                            s_p[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>29)
        {
                if (blockIdx.x<gridDim.x-1)     s_p[threadIdx.x+3][threadIdx.y]=
                    p[id+2];
                else                            s_p[threadIdx.x+3][threadIdx.y
                    ]=0.0f;
        }
        __syncthreads();

```

```
    float diff1=1.125f*(s_p[threadIdx.x+2][threadIdx.y]-s_p[threadIdx.x+1][threadIdx.y])
                -0.041666666666667f*(s_p[threadIdx.x+3][threadIdx.y]-s_p[threadIdx.x][threadIdx.y]);
    convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
    vz[id]+=convpz[ik];
}
__global__ void cuda_PML_vx_4(float *p,  float *convpx, float *bx, float *vx, float _dx, int npml, int nnz, int nnx)
{
    // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
    // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
    // only 2 blocks used vertically, blockIdx.y=0, 1

    // id: position in whole zone(including PML)
    // ik: position in top or bottom PML zone itself
    // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
    int i1=threadIdx.x+blockIdx.x*blockDim.x;
    int i2=threadIdx.y+blockIdx.y*npml;
    int ik=i1+i2*nnz;
    int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

    __shared__ float s_p[Block_Size1][35];// npml+3=35; Block_SizeX=32; Block_SizeY=8;
    s_p[threadIdx.x][threadIdx.y+1]=p[id];
    if (threadIdx.y<1)
    {
        if (blockIdx.y)             s_p[threadIdx.x][threadIdx.y]=p[id-nnz];
        else                        s_p[threadIdx.x][threadIdx.y]=0.0f;
    }
    if (threadIdx.y>29)
    {
        if (blockIdx.y<gridDim.y-1) s_p[threadIdx.x][threadIdx.y+3]=p[id+2*nnz];
        else                        s_p[threadIdx.x][threadIdx.y+3]=0.0f;
    }
    __syncthreads();

    float diff2=1.125f*(s_p[threadIdx.x][threadIdx.y+2]-s_p[threadIdx.x][threadIdx.y+1])
                -0.041666666666667f*(s_p[threadIdx.x][threadIdx.y+3]-s_p[threadIdx.x][threadIdx.y]);
    convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
    vx[id]+=convpx[ik];
}

__global__ void cuda_forward_up_4(float *up, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
    int i1=blockIdx.x*blockDim.x+threadIdx.x;
    int i2=blockIdx.y*blockDim.y+threadIdx.y;
    int id=i1+i2*nnz;

    __shared__ float s_vx[Block_Size1][Block_Size2+3];
    __shared__ float s_vz[Block_Size1+3][Block_Size2];
```

```
        s_vx[threadIdx.x][threadIdx.y+2]=vx[id];
        s_vz[threadIdx.x+2][threadIdx.y]=vz[id];

        if (threadIdx.x<2)
        {
                if (blockIdx.x)                 s_vz[threadIdx.x][threadIdx.y]=vz[id-2];
                else                            s_vz[threadIdx.x][threadIdx.y]=0.0f;
        }
        if (threadIdx.x>blockDim.x-2)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+3][threadIdx.y]=vz[id+1];
                else                            s_vz[threadIdx.x+3][threadIdx.y]=0.0f;
        }
        if (threadIdx.y<2)
        {
                if (blockIdx.y)                 s_vx[threadIdx.x][threadIdx.y]=vx[id-2*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y]=0.0f;
        }
        if (threadIdx.y>blockDim.y-2)
        {
                if (blockIdx.y<gridDim.y-1)     s_vx[threadIdx.x][threadIdx.y+3]=vx[id+nnz];
                else                            s_vx[threadIdx.x][threadIdx.y+3]=0.0f;
        }
        __syncthreads();

        float diff2=1.125f*(s_vx[threadIdx.x][threadIdx.y+2]-s_vx[threadIdx.x][threadIdx.y+1])
                        -0.041666666666667f*(s_vx[threadIdx.x][threadIdx.y+3]-s_vx[threadIdx.x][threadIdx.y]);
        float diff1=1.125f*(s_vz[threadIdx.x+2][threadIdx.y]-s_vz[threadIdx.x+1][threadIdx.y])+
                        -0.041666666666667f*(s_vz[threadIdx.x+3][threadIdx.y]-s_vz[threadIdx.x][threadIdx.y]);
        up[id]=_dz*diff1+_dx*diff2;
}

__global__ void cuda_PML_upz_4(float *up,  float *convvz, float *bz, float *vz, float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

```

```
        __shared__ float s_vz[35][Block_Size2];
        s_vz[threadIdx.x+2][threadIdx.y]=vz[id];
        if (threadIdx.x<2)
        {
                if (blockIdx.x)             s_vz[threadIdx.x][threadIdx.y]=
                    vz[id-2];
                else                        s_vz[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>30)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+3][threadIdx.y
                    ]=vz[id+1];
                else                        s_vz[threadIdx.x+3][threadIdx.y
                    ]=0.0f;
        }
        __syncthreads();

        float diff1=1.125f*(s_vz[threadIdx.x+2][threadIdx.y]-s_vz[threadIdx.x
            +1][threadIdx.y])+
                        -0.041666666666667f*(s_vz[threadIdx.x+3][threadIdx.y]-
                            s_vz[threadIdx.x][threadIdx.y]);
        convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        up[id]+=convvz[ik];
}
__global__ void cuda_PML_upx_4(float *up, float *convvx, float *bx, float *vx,
    float _dx, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+i2*nnz;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

        __shared__ float s_vx[Block_Size1][35];
        s_vx[threadIdx.x][threadIdx.y+2]=vx[id];
        if (threadIdx.y<2)
        {
                if (blockIdx.y)             s_vx[threadIdx.x][threadIdx.y]=
                    vx[id-2*nnz];
                else                        s_vx[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>30)
        {
                if (blockIdx.y<gridDim.y-1)     s_vx[threadIdx.x][threadIdx.y
                    +3]=vx[id+nnz];
                else                        s_vx[threadIdx.x][threadIdx.y
                    +3]=0.0f;
        }
        __syncthreads();

```

```
	float diff2=1.125f*(s_vx[threadIdx.x][threadIdx.y+2]-s_vx[threadIdx.x][threadIdx.y+1])
			-0.041666666666667f*(s_vx[threadIdx.x][threadIdx.y+3]-s_vx[threadIdx.x][threadIdx.y]);
	convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
	up[id]+=convvx[ik];
}


//============================================= NJ=6 ==========================================================
__global__ void cuda_forward_v_6(float *p, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
	int i1=blockIdx.x*blockDim.x+threadIdx.x;
	int i2=blockIdx.y*blockDim.y+threadIdx.y;
	int id=i1+i2*nnz;

	__shared__ float s_p[Block_Size1+5][Block_Size2+5];
	s_p[threadIdx.x+2][threadIdx.y+2]=p[id];
	if (threadIdx.x<2)
	{
		if (blockIdx.x)			s_p[threadIdx.x][threadIdx.y+2]=p[id-2];
		else				s_p[threadIdx.x][threadIdx.y+2]=0.0f;
	}
	if (threadIdx.x>blockDim.x-4)
	{
		if (blockIdx.x<gridDim.x-1)	s_p[threadIdx.x+5][threadIdx.y+2]=p[id+3];
		else				s_p[threadIdx.x+5][threadIdx.y+2]=0.0f;
	}
	if (threadIdx.y<2)
	{
		if (blockIdx.y)			s_p[threadIdx.x+2][threadIdx.y]=p[id-2*nnz];
		else				s_p[threadIdx.x+2][threadIdx.y]=0.0f;
	}
	if (threadIdx.y>blockDim.y-4)
	{
		if (blockIdx.y<gridDim.y-1)	s_p[threadIdx.x+2][threadIdx.y+5]=p[id+3*nnz];
		else				s_p[threadIdx.x+2][threadIdx.y+5]=0.0f;
	}
	__syncthreads();

	float diff1=1.171875f*(s_p[threadIdx.x+3][threadIdx.y+2]-s_p[threadIdx.x+2][threadIdx.y+2])
			-0.065104166666667f*(s_p[threadIdx.x+4][threadIdx.y+2]-s_p[threadIdx.x+1][threadIdx.y+2])
			+0.0046875f*(s_p[threadIdx.x+5][threadIdx.y+2]-s_p[threadIdx.x][threadIdx.y+2]);
	float diff2=1.171875f*(s_p[threadIdx.x+2][threadIdx.y+3]-s_p[threadIdx.x+2][threadIdx.y+2])
			-0.065104166666667f*(s_p[threadIdx.x+2][threadIdx.y+4]-s_p[threadIdx.x+2][threadIdx.y+1])
```

```
                        +0.0046875f*(s_p[threadIdx.x+2][threadIdx.y+5]-s_p[threadIdx.x+2][threadIdx.y]);
        vz[id]=_dz*diff1;
        vx[id]=_dx*diff2;
}

__global__ void cuda_PML_vz_6(float *p,  float *convpz, float *bz, float *vz, float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

        __shared__ float s_p[37][Block_Size2];
        s_p[threadIdx.x+2][threadIdx.y]=p[id];
        if (threadIdx.x<2)
        {
                if (blockIdx.x)                 s_p[threadIdx.x][threadIdx.y]=p[id-2];
                else                            s_p[threadIdx.x][threadIdx.y]=0.0f;
        }
        if (threadIdx.x>28)
        {
                if (blockIdx.x<gridDim.x-1)     s_p[threadIdx.x+5][threadIdx.y]=p[id+3];
                else                            s_p[threadIdx.x+5][threadIdx.y]=0.0f;
        }
        __syncthreads();

        float diff1=1.171875f*(s_p[threadIdx.x+3][threadIdx.y]-s_p[threadIdx.x+2][threadIdx.y])
                        -0.065104166666667f*(s_p[threadIdx.x+4][threadIdx.y]-s_p[threadIdx.x+1][threadIdx.y])
                        +0.0046875f*(s_p[threadIdx.x+5][threadIdx.y]-s_p[threadIdx.x][threadIdx.y]);
        convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        vz[id]+=convpz[ik];
}
__global__ void cuda_PML_vx_6(float *p,  float *convpx, float *bx, float *vx, float _dx, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
```

```
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+i2*nnz;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

        __shared__ float s_p[Block_Size1][37];
        s_p[threadIdx.x][threadIdx.y+2]=p[id];
        if (threadIdx.y<2)
        {
                if (blockIdx.y)                 s_p[threadIdx.x][threadIdx.y]=p[
                    id-2*nnz];
                else                            s_p[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>28)
        {
                if (blockIdx.y<gridDim.y-1)     s_p[threadIdx.x][threadIdx.y+5]=
                    p[id+3*nnz];
                else                            s_p[threadIdx.x][threadIdx.y
                    +5]=0.0f;
        }
        __syncthreads();

        float diff2=1.171875f*(s_p[threadIdx.x][threadIdx.y+3]-s_p[threadIdx.x][
            threadIdx.y+2])
                        -0.065104166666667f*(s_p[threadIdx.x][threadIdx.y+4]-s_p
                            [threadIdx.x][threadIdx.y+1])
                        +0.0046875f*(s_p[threadIdx.x][threadIdx.y+5]-s_p[
                            threadIdx.x][threadIdx.y]);
        convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
        vx[id]+=convpx[ik];
}
__global__ void cuda_forward_up_6(float *up, float *vx, float *vz, float _dx,
    float _dz, int npml, int nnz, int nnx)
{
        int i1=blockIdx.x*blockDim.x+threadIdx.x;
        int i2=blockIdx.y*blockDim.y+threadIdx.y;
        int id=i1+i2*nnz;

        __shared__ float s_vx[Block_Size1][Block_Size2+5];
        __shared__ float s_vz[Block_Size1+5][Block_Size2];
        s_vx[threadIdx.x][threadIdx.y+3]=vx[id];
        s_vz[threadIdx.x+3][threadIdx.y]=vz[id];

        if (threadIdx.x<3)
        {
                if (blockIdx.x)                 s_vz[threadIdx.x][threadIdx.y]=
                    vz[id-3];
                else                            s_vz[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>blockDim.x-3)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+5][threadIdx.y
                    ]=vz[id+2];
                else                            s_vz[threadIdx.x+5][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y<3)
        {
```

```
                if (blockIdx.y)                 s_vx[threadIdx.x][threadIdx.y]=
                    vx[id-3*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>blockDim.y-3)
        {
                if (blockIdx.y<gridDim.y-1)     s_vx[threadIdx.x][threadIdx.y
                    +5]=vx[id+2*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y
                    +5]=0.0f;
        }
        __syncthreads();


        float diff2=1.171875f*(s_vx[threadIdx.x][threadIdx.y+3]-s_vx[threadIdx.x
            ][threadIdx.y+2])
                                -0.065104166666667f*(s_vx[threadIdx.x][threadIdx.y+4]-
                                    s_vx[threadIdx.x][threadIdx.y+1])+
                                0.0046875f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
                                    threadIdx.x][threadIdx.y]);
        float diff1=1.171875f*(s_vz[threadIdx.x+3][threadIdx.y]-s_vz[threadIdx.x
            +2][threadIdx.y])+
                                -0.065104166666667f*(s_vz[threadIdx.x+4][threadIdx.y]-
                                    s_vz[threadIdx.x+1][threadIdx.y])+
                                0.0046875f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
                                    threadIdx.x][threadIdx.y]);
        up[id]=(_dz*diff1+_dx*diff2);
}

__global__ void cuda_PML_upz_6(float *up,  float *convvz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

        __shared__ float s_vz[37][Block_Size2];// npml+5=37; Block_SizeX=32;
            Block_SizeY=8;
        s_vz[threadIdx.x+3][threadIdx.y]=vz[id];
        if (threadIdx.x<3)
        {
                if (blockIdx.x)                 s_vz[threadIdx.x][threadIdx.y]=
                    vz[id-3];
                else                            s_vz[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>29)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+5][threadIdx.y
                    ]=vz[id+2];
```

```
                else                                s_vz[threadIdx.x+5][threadIdx.y]=0.0f;
        }
        __syncthreads();

        float diff1=1.171875f*(s_vz[threadIdx.x+3][threadIdx.y]-s_vz[threadIdx.x+2][threadIdx.y])
                        -0.065104166666667f*(s_vz[threadIdx.x+4][threadIdx.y]-s_vz[threadIdx.x+1][threadIdx.y])
                        +0.0046875f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[threadIdx.x][threadIdx.y]);
        convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        up[id]+=convvz[ik];
}
__global__ void cuda_PML_upx_6(float *up, float *convvx, float *bx, float *vx, float _dx, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+i2*nnz;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

        __shared__ float s_vx[Block_Size1][37];
        s_vx[threadIdx.x][threadIdx.y+3]=vx[id];
        if (threadIdx.y<3)
        {
                if (blockIdx.y)                     s_vx[threadIdx.x][threadIdx.y]=vx[id-3*nnz];
                else                                s_vx[threadIdx.x][threadIdx.y]=0.0f;
        }
        if (threadIdx.y>29)
        {
                if (blockIdx.y<gridDim.y-1)         s_vx[threadIdx.x][threadIdx.y+5]=vx[id+2*nnz];
                else                                s_vx[threadIdx.x][threadIdx.y+5]=0.0f;
        }
        __syncthreads();

        float diff2=1.171875f*(s_vx[threadIdx.x][threadIdx.y+3]-s_vx[threadIdx.x][threadIdx.y+2])
                        -0.065104166666667f*(s_vx[threadIdx.x][threadIdx.y+4]-s_vx[threadIdx.x][threadIdx.y+1])
                        +0.0046875f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[threadIdx.x][threadIdx.y]);
        convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
        up[id]+=convvx[ik];
}
```

```
//============================================ NJ=8 ============================================
__global__ void cuda_forward_v_8(float *p, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
	int i1=blockIdx.x*blockDim.x+threadIdx.x;
	int i2=blockIdx.y*blockDim.y+threadIdx.y;
	int id=i1+nnz*i2;

	__shared__ float s_p[Block_Size1+7][Block_Size2+7];
	s_p[threadIdx.x+3][threadIdx.y+3]=p[id];
	if (threadIdx.x<3)
	{
		if (blockIdx.x)			s_p[threadIdx.x][threadIdx.y+3]=p[id-3];
		else				s_p[threadIdx.x][threadIdx.y+3]=0.0f;
	}
	if (threadIdx.x>blockDim.x-5)
	{
		if (blockIdx.x<gridDim.x-1)	s_p[threadIdx.x+7][threadIdx.y+3]=p[id+4];
		else				s_p[threadIdx.x+7][threadIdx.y+3]=0.0f;
	}
	if (threadIdx.y<3)
	{
		if (blockIdx.y)			s_p[threadIdx.x+3][threadIdx.y]=p[id-3*nnz];
		else				s_p[threadIdx.x+3][threadIdx.y]=0.0f;
	}
	if (threadIdx.y>blockDim.y-5)
	{
		if (blockIdx.y<gridDim.y-1)	s_p[threadIdx.x+3][threadIdx.y+7]=p[id+4*nnz];
		else				s_p[threadIdx.x+3][threadIdx.y+7]=0.0f;
	}
	__syncthreads();

	float diff1=1.1962890625000f*(s_p[threadIdx.x+4][threadIdx.y+3]-s_p[threadIdx.x+3][threadIdx.y+3])
			-0.0797526041667f*(s_p[threadIdx.x+5][threadIdx.y+3]-s_p[threadIdx.x+2][threadIdx.y+3])
			+0.0095703125000f*(s_p[threadIdx.x+6][threadIdx.y+3]-s_p[threadIdx.x+1][threadIdx.y+3])
			-0.0006975446429f*(s_p[threadIdx.x+7][threadIdx.y+3]-s_p[threadIdx.x][threadIdx.y+3]);
	float diff2=1.1962890625000f*(s_p[threadIdx.x+3][threadIdx.y+4]-s_p[threadIdx.x+3][threadIdx.y+3])
			-0.0797526041667f*(s_p[threadIdx.x+3][threadIdx.y+5]-s_p[threadIdx.x+3][threadIdx.y+2])
			+0.0095703125000f*(s_p[threadIdx.x+3][threadIdx.y+6]-s_p[threadIdx.x+3][threadIdx.y+1])
			-0.0006975446429f*(s_p[threadIdx.x+3][threadIdx.y+7]-s_p[threadIdx.x+3][threadIdx.y]);

	vz[id]=_dz*diff1;
	vx[id]=_dx*diff2;
}
```

```

__global__ void cuda_PML_vz_8(float *p, float *convpz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
	// bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
	    direction
	// bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
	    direction
	// only 2 blocks used vertically, blockIdx.y=0, 1

	// id: position in whole zone(including PML)
	// ik: position in top or bottom PML zone itself
	// blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
	int i1=threadIdx.x+blockIdx.x*npml;
	int i2=threadIdx.y+blockIdx.y*blockDim.y;
	int ik=i1+2*npml*i2;
	int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

	__shared__ float s_p[39][Block_Size2];
	s_p[threadIdx.x+3][threadIdx.y]=p[id];
	if (threadIdx.x<3)
	{
		if (blockIdx.x)			s_p[threadIdx.x][threadIdx.y]=p[
		    id-3];
		else				s_p[threadIdx.x][threadIdx.y
		    ]=0.0f;
	}
	if (threadIdx.x>27)
	{
		if (blockIdx.x<gridDim.x-1)	s_p[threadIdx.x+7][threadIdx.y]=
		    p[id+4];
		else				s_p[threadIdx.x+7][threadIdx.y
		    ]=0.0f;
	}
	__syncthreads();

	float diff1=1.1962890625000f*(s_p[threadIdx.x+4][threadIdx.y]-s_p[
	    threadIdx.x+3][threadIdx.y])
			-0.0797526041667f*(s_p[threadIdx.x+5][threadIdx.y]-s_p[
			    threadIdx.x+2][threadIdx.y])
			+0.0095703125000f*(s_p[threadIdx.x+6][threadIdx.y]-s_p[
			    threadIdx.x+1][threadIdx.y])
			-0.0006975446429f*(s_p[threadIdx.x+7][threadIdx.y]-s_p[
			    threadIdx.x][threadIdx.y]);
	convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
	vz[id]+=convpz[ik];
}
__global__ void cuda_PML_vx_8(float *p, float *convpx, float *bx, float *vx,
    float _dx, int npml, int nnz, int nnx)
{
	// bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
	    direction
	// bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
	    direction
	// only 2 blocks used vertically, blockIdx.y=0, 1

	// id: position in whole zone(including PML)
	// ik: position in top or bottom PML zone itself
	// blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
	int i1=threadIdx.x+blockIdx.x*blockDim.x;
```

```
	int i2=threadIdx.y+blockIdx.y*npml;
	int ik=i1+i2*nnz;
	int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

	__shared__ float s_p[Block_Size1][39];// npml+7=39; Block_SizeX=32; Block_SizeY=8;
	s_p[threadIdx.x][threadIdx.y+3]=p[id];
	if (threadIdx.y<3)
	{
		if (blockIdx.y)			s_p[threadIdx.x][threadIdx.y]=p[id-3*nnz];
		else				s_p[threadIdx.x][threadIdx.y]=0.0f;
	}
	if (threadIdx.y>27)//32-4
	{
		if (blockIdx.y<gridDim.y-1)	s_p[threadIdx.x][threadIdx.y+7]=p[id+4*nnz];
		else				s_p[threadIdx.x][threadIdx.y+7]=0.0f;
	}
	__syncthreads();

	float diff2=1.1962890625000f*(s_p[threadIdx.x][threadIdx.y+4]-s_p[threadIdx.x][threadIdx.y+3])
			-0.0797526041667f*(s_p[threadIdx.x][threadIdx.y+5]-s_p[threadIdx.x][threadIdx.y+2])+
			0.0095703125000f*(s_p[threadIdx.x][threadIdx.y+6]-s_p[threadIdx.x][threadIdx.y+1])+
			-0.0006975446429f*(s_p[threadIdx.x][threadIdx.y+7]-s_p[threadIdx.x][threadIdx.y]);
	convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
	vx[id]+=convpx[ik];
}

__global__ void cuda_forward_up_8(float *up, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
	int i1=blockIdx.x*blockDim.x+threadIdx.x;
	int i2=blockIdx.y*blockDim.y+threadIdx.y;
	int id=i1+i2*nnz;

	__shared__ float s_vx[Block_Size1][Block_Size2+7];
	__shared__ float s_vz[Block_Size1+7][Block_Size2];
	s_vx[threadIdx.x][threadIdx.y+4]=vx[id];
	s_vz[threadIdx.x+4][threadIdx.y]=vz[id];

	if (threadIdx.x<4)
	{
		if (blockIdx.x)			s_vz[threadIdx.x][threadIdx.y]=vz[id-4];
		else				s_vz[threadIdx.x][threadIdx.y]=0.0f;
	}
	if (threadIdx.x>blockDim.x-4)
	{
		if (blockIdx.x<gridDim.x-1)	s_vz[threadIdx.x+7][threadIdx.y]=vz[id+3];
		else				s_vz[threadIdx.x+7][threadIdx.y]=0.0f;
	}
```

```
        if (threadIdx.y<4)
        {
                if (blockIdx.y)                 s_vx[threadIdx.x][threadIdx.y]=
                    vx[id-4*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>blockDim.y-4)
        {
                if (blockIdx.y<gridDim.y-1)     s_vx[threadIdx.x][threadIdx.y
                    +7]=vx[id+3*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y
                    +7]=0.0f;
        }
        __syncthreads();


        float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+4]-s_vx[
            threadIdx.x][threadIdx.y+3])
                        -0.0797526041667f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx
                            [threadIdx.x][threadIdx.y+2])+
                        0.0095703125000f*(s_vx[threadIdx.x][threadIdx.y+6]-s_vx[
                            threadIdx.x][threadIdx.y+1])+
                        -0.0006975446429f*(s_vx[threadIdx.x][threadIdx.y+7]-s_vx
                            [threadIdx.x][threadIdx.y]);
        float diff1=1.1962890625000f*(s_vz[threadIdx.x+4][threadIdx.y]-s_vz[
            threadIdx.x+3][threadIdx.y])+
                        -0.0797526041667f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz
                            [threadIdx.x+2][threadIdx.y])+
                        0.0095703125000f*(s_vz[threadIdx.x+6][threadIdx.y]-s_vz[
                            threadIdx.x+1][threadIdx.y])+
                        -0.0006975446429f*(s_vz[threadIdx.x+7][threadIdx.y]-s_vz
                            [threadIdx.x][threadIdx.y]);
        up[id]=_dz*diff1+_dx*diff2;
}
__global__ void cuda_PML_upz_8(float *up, float *convvz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

        __shared__ float s_vz[39][Block_Size2];
        s_vz[threadIdx.x+4][threadIdx.y]=vz[id];
        if (threadIdx.x<4)
        {
                if (blockIdx.x)                 s_vz[threadIdx.x][threadIdx.y]=
                    vz[id-4];
                else                            s_vz[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
```

```
        if (threadIdx.x>28)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+7][threadIdx.y]=vz[id+3];
                else                            s_vz[threadIdx.x+7][threadIdx.y]=0.0f;
        }
        __syncthreads();

        float diff1=1.1962890625000f*(s_vz[threadIdx.x+4][threadIdx.y]-s_vz[threadIdx.x+3][threadIdx.y])
                        -0.0797526041667f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[threadIdx.x+2][threadIdx.y])
                        +0.0095703125000f*(s_vz[threadIdx.x+6][threadIdx.y]-s_vz[threadIdx.x+1][threadIdx.y])
                        -0.0006975446429f*(s_vz[threadIdx.x+7][threadIdx.y]-s_vz[threadIdx.x][threadIdx.y]);
        convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        up[id]+=convvz[ik];
}
__global__ void cuda_PML_upx_8(float *up, float *convvx, float *bx, float *vx, float _dx, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+i2*nnz;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

        __shared__ float s_vx[Block_Size1][39];
        s_vx[threadIdx.x][threadIdx.y+4]=vx[id];
        if (threadIdx.y<4)
        {
                if (blockIdx.y)                 s_vx[threadIdx.x][threadIdx.y]=vx[id-4*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y]=0.0f;
        }
        if (threadIdx.y>28)
        {
                if (blockIdx.y<gridDim.y-1)     s_vx[threadIdx.x][threadIdx.y+7]=vx[id+3*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y+7]=0.0f;
        }
        __syncthreads();

        float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+4]-s_vx[threadIdx.x][threadIdx.y+3])
                        -0.0797526041667f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[threadIdx.x][threadIdx.y+2])
                        +0.0095703125000f*(s_vx[threadIdx.x][threadIdx.y+6]-s_vx[threadIdx.x][threadIdx.y+1])
```

```
                                        -0.0006975446429f*(s_vx[threadIdx.x][threadIdx.y+7]-s_vx
                                            [threadIdx.x][threadIdx.y]);
          convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
          up[id]+=convvx[ik];
}


//============================================================ NJ=10
    =================================================================
__global__ void cuda_forward_v_10(float *p, float *vx, float *vz, float _dx,
    float _dz, int npml, int nnz, int nnx)
{
          int i1=blockIdx.x*blockDim.x+threadIdx.x;
          int i2=blockIdx.y*blockDim.y+threadIdx.y;
          int id=i1+i2*nnz;

          __shared__ float s_p[Block_Size1+9][Block_Size2+9];
          s_p[threadIdx.x+4][threadIdx.y+4]=p[id];
          if (threadIdx.x<4)
          {
                    if (blockIdx.x)                s_p[threadIdx.x][threadIdx.y+4]=
                        p[id-4];
                    else                           s_p[threadIdx.x][threadIdx.y
                        +4]=0.0f;
          }
          if (threadIdx.x>blockDim.x-6)
          {
                    if (blockIdx.x<gridDim.x-1)    s_p[threadIdx.x+9][threadIdx.y
                        +4]=p[id+5];
                    else                           s_p[threadIdx.x+9][threadIdx.y
                        +4]=0.0f;
          }
          if (threadIdx.y<4)
          {
                    if (blockIdx.y)                s_p[threadIdx.x+4][threadIdx.y]=
                        p[id-4*nnz];
                    else                           s_p[threadIdx.x+4][threadIdx.y
                        ]=0.0f;
          }
          if (threadIdx.y>blockDim.y-6)
          {
                    if (blockIdx.y<gridDim.y-1)    s_p[threadIdx.x+4][threadIdx.y
                        +9]=p[id+5*nnz];
                    else                           s_p[threadIdx.x+4][threadIdx.y
                        +9]=0.0f;
          }
          __syncthreads();

          float diff1=1.211242675781250f*(s_p[threadIdx.x+5][threadIdx.y+4]-s_p[
              threadIdx.x+4][threadIdx.y+4])
                                        -0.089721679687500f*(s_p[threadIdx.x+6][threadIdx.y+4]-
                                            s_p[threadIdx.x+3][threadIdx.y+4])
                                        +0.013842773437500f*(s_p[threadIdx.x+7][threadIdx.y+4]-
                                            s_p[threadIdx.x+2][threadIdx.y+4])
                                        -0.001765659877232f*(s_p[threadIdx.x+8][threadIdx.y+4]-
                                            s_p[threadIdx.x+1][threadIdx.y+4])
                                        +0.000118679470486f*(s_p[threadIdx.x+9][threadIdx.y+4]-
                                            s_p[threadIdx.x][threadIdx.y+4]);
          float diff2= 1.211242675781250f*(s_p[threadIdx.x+4][threadIdx.y+5]-s_p[
              threadIdx.x+4][threadIdx.y+4])
```

```
                                -0.089721679687500f*(s_p[threadIdx.x+4][threadIdx.y+6]-
                                    s_p[threadIdx.x+4][threadIdx.y+3])
                                +0.013842773437500f*(s_p[threadIdx.x+4][threadIdx.y+7]-
                                    s_p[threadIdx.x+4][threadIdx.y+2])
                                -0.001765659877232f*(s_p[threadIdx.x+4][threadIdx.y+8]-
                                    s_p[threadIdx.x+4][threadIdx.y+1])
                                +0.000118679470486f*(s_p[threadIdx.x+4][threadIdx.y+9]-
                                    s_p[threadIdx.x+4][threadIdx.y]);
        vz[id]=_dz*diff1;
        vx[id]=_dx*diff2;
}

__global__ void cuda_PML_vz_10(float *p, float *convpz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

        __shared__ float s_p[41][Block_Size2];
        s_p[threadIdx.x+4][threadIdx.y]=p[id];
        if (threadIdx.x<4)
        {
                if (blockIdx.x)                 s_p[threadIdx.x][threadIdx.y]=p[
                    id-4];
                else                            s_p[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>26)
        {
                if (blockIdx.x<gridDim.x-1)     s_p[threadIdx.x+9][threadIdx.y]=
                    p[id+5];
                else                            s_p[threadIdx.x+9][threadIdx.y
                    ]=0.0f;
        }
        __syncthreads();

        float diff1=1.1962890625000f*(s_p[threadIdx.x+5][threadIdx.y]-s_p[
            threadIdx.x+4][threadIdx.y])
                                -0.089721679687500f*(s_p[threadIdx.x+6][threadIdx.y]-s_p
                                    [threadIdx.x+3][threadIdx.y])
                                +0.013842773437500f*(s_p[threadIdx.x+7][threadIdx.y]-s_p
                                    [threadIdx.x+2][threadIdx.y])
                                -0.001765659877232f*(s_p[threadIdx.x+8][threadIdx.y]-s_p
                                    [threadIdx.x+1][threadIdx.y])
                                +0.000118679470486f*(s_p[threadIdx.x+9][threadIdx.y]-s_p
                                    [threadIdx.x][threadIdx.y]);
        convpz[ik]=bz[ik]*convpz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        vz[id]+=convpz[ik];
}
```

```
__global__ void cuda_PML_vx_10(float *p, float *convpx, float *bx, float *vx, float _dx, int npml, int nnz, int nnx)
{
	// bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z direction
	// bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z direction
	// only 2 blocks used vertically, blockIdx.y=0, 1

	// id: position in whole zone(including PML)
	// ik: position in top or bottom PML zone itself
	// blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
	int i1=threadIdx.x+blockIdx.x*blockDim.x;
	int i2=threadIdx.y+blockIdx.y*npml;
	int ik=i1+i2*nnz;
	int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);
	__shared__ float s_p[Block_Size1][41];
	s_p[threadIdx.x][threadIdx.y+4]=p[id];
	if (threadIdx.y<4)
	{
		if (blockIdx.y)			s_p[threadIdx.x][threadIdx.y]=p[id-4*nnz];
		else				s_p[threadIdx.x][threadIdx.y]=0.0f;
	}
	if (threadIdx.y>26)
	{
		if (blockIdx.y<gridDim.y-1)	s_p[threadIdx.x][threadIdx.y+9]=p[id+5*nnz];
		else				s_p[threadIdx.x][threadIdx.y+9]=0.0f;
	}
	__syncthreads();

	float diff2=1.196289062500f*(s_p[threadIdx.x][threadIdx.y+5]-s_p[threadIdx.x][threadIdx.y+4])
			-0.089721679687500f*(s_p[threadIdx.x][threadIdx.y+6]-s_p[threadIdx.x][threadIdx.y+3])
			+0.013842773437500f*(s_p[threadIdx.x][threadIdx.y+7]-s_p[threadIdx.x][threadIdx.y+2])
			-0.001765659877232f*(s_p[threadIdx.x][threadIdx.y+8]-s_p[threadIdx.x][threadIdx.y+1])
			+0.000118679470486f*(s_p[threadIdx.x][threadIdx.y+9]-s_p[threadIdx.x][threadIdx.y]);
	convpx[ik]=bx[ik]*convpx[ik]+(bx[ik]-1.0f)*_dx*diff2;
	vx[id]+=convpx[ik];
}

__global__ void cuda_forward_up_10(float *up, float *vx, float *vz, float _dx, float _dz, int npml, int nnz, int nnx)
{
	int i1=blockIdx.x*blockDim.x+threadIdx.x;
	int i2=blockIdx.y*blockDim.y+threadIdx.y;
	int id=i1+i2*nnz;

	__shared__ float s_vx[Block_Size1][Block_Size2+9];
	__shared__ float s_vz[Block_Size1+9][Block_Size2];
	s_vx[threadIdx.x][threadIdx.y+5]=vx[id];
	s_vz[threadIdx.x+5][threadIdx.y]=vz[id];

	if (threadIdx.x<5)
```

```
        {
                if (blockIdx.x)                 s_vz[threadIdx.x][threadIdx.y]=
                    vz[id-5];
                else                            s_vz[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>blockDim.x-5)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+9][threadIdx.y
                    ]=vz[id+4];
                else                            s_vz[threadIdx.x+9][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y<5)
        {
                if (blockIdx.y)                 s_vx[threadIdx.x][threadIdx.y]=
                    vx[id-5*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.y>blockDim.y-5)
        {
                if (blockIdx.y<gridDim.y-1)     s_vx[threadIdx.x][threadIdx.y
                    +9]=vx[id+4*nnz];
                else                            s_vx[threadIdx.x][threadIdx.y
                    +9]=0.0f;
        }
        __syncthreads();

        float diff1=1.1962890625000f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
            threadIdx.x+4][threadIdx.y])
                        -0.0897216796875 00f*(s_vz[threadIdx.x+6][threadIdx.y]-
                            s_vz[threadIdx.x+3][threadIdx.y])
                        +0.0138427734375 00f*(s_vz[threadIdx.x+7][threadIdx.y]-
                            s_vz[threadIdx.x+2][threadIdx.y])
                        -0.001765659877232f*(s_vz[threadIdx.x+8][threadIdx.y]-
                            s_vz[threadIdx.x+1][threadIdx.y])
                        +0.000118679470486f*(s_vz[threadIdx.x+9][threadIdx.y]-
                            s_vz[threadIdx.x][threadIdx.y]);
        float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
            threadIdx.x][threadIdx.y+4])
                        -0.089721679687500f*(s_vx[threadIdx.x][threadIdx.y+6]-
                            s_vx[threadIdx.x][threadIdx.y+3])
                        +0.013842773437500f*(s_vx[threadIdx.x][threadIdx.y+7]-
                            s_vx[threadIdx.x][threadIdx.y+2])
                        -0.001765659877232f*(s_vx[threadIdx.x][threadIdx.y+8]-
                            s_vx[threadIdx.x][threadIdx.y+1])
                        +0.000118679470486f*(s_vx[threadIdx.x][threadIdx.y+9]-
                            s_vx[threadIdx.x][threadIdx.y]);
        up[id]=(_dz*diff1+_dx*diff2);
}
__global__ void cuda_PML_upz_10(float *up, float *convvz, float *bz, float *vz,
    float _dz, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
```

```
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*npml;
        int i2=threadIdx.y+blockIdx.y*blockDim.y;
        int ik=i1+2*npml*i2;
        int id=blockIdx.x*(nnz-npml)+threadIdx.x+nnz*i2;

        __shared__ float s_vz[41][Block_Size2];
        s_vz[threadIdx.x+5][threadIdx.y]=vz[id];
        if (threadIdx.x<5)
        {
                if (blockIdx.x)                 s_vz[threadIdx.x][threadIdx.y]=
                    vz[id-5];
                else                            s_vz[threadIdx.x][threadIdx.y
                    ]=0.0f;
        }
        if (threadIdx.x>27)
        {
                if (blockIdx.x<gridDim.x-1)     s_vz[threadIdx.x+9][threadIdx.y
                    ]=vz[id+4];
                else                            s_vz[threadIdx.x+9][threadIdx.y
                    ]=0.0f;
        }
        __syncthreads();


        float diff1=1.1962890625000f*(s_vz[threadIdx.x+5][threadIdx.y]-s_vz[
            threadIdx.x+4][threadIdx.y])
                        -0.0897216796875 00f*(s_vz[threadIdx.x+6][threadIdx.y]-
                            s_vz[threadIdx.x+3][threadIdx.y])
                        +0.013842773437500f*(s_vz[threadIdx.x+7][threadIdx.y]-
                            s_vz[threadIdx.x+2][threadIdx.y])
                        -0.001765659877232f*(s_vz[threadIdx.x+8][threadIdx.y]-
                            s_vz[threadIdx.x+1][threadIdx.y])
                        +0.000118679470486f*(s_vz[threadIdx.x+9][threadIdx.y]-
                            s_vz[threadIdx.x][threadIdx.y]);
        convvz[ik]=bz[ik]*convvz[ik]+(bz[ik]-1.0f)*_dz*diff1;
        up[id]+=convvz[ik];
}
__global__ void cuda_PML_upx_10(float *up, float *convvx, float *bx, float *vx,
    float _dx, int npml, int nnz, int nnx)
{
        // bz1: top and bottom PML ABC coefficients, decay p (px,pz) along z
            direction
        // bz2: top and bottom PML ABC coefficients, decay v (vx,vz) along z
            direction
        // only 2 blocks used vertically, blockIdx.y=0, 1

        // id: position in whole zone(including PML)
        // ik: position in top or bottom PML zone itself
        // blockIdx.y==0, top PML zone; blockIdx.y==1, bottom PML zone
        int i1=threadIdx.x+blockIdx.x*blockDim.x;
        int i2=threadIdx.y+blockIdx.y*npml;
        int ik=i1+i2*nnz;
        int id=i1+nnz*(blockIdx.y*(nnx-npml)+threadIdx.y);

        __shared__ float s_vx[Block_Size1][41];
        s_vx[threadIdx.x][threadIdx.y+5]=vx[id];
        if (threadIdx.y<5)
        {
```

```
                if (blockIdx.y)                         s_vx[threadIdx.x][threadIdx.y]=
                   vx[id-5*nnz];
                else                                    s_vx[threadIdx.x][threadIdx.y
                   ]=0.0f;
        }
        if (threadIdx.y>27)
        {
                if (blockIdx.y<gridDim.y-1)             s_vx[threadIdx.x][threadIdx.y
                   +9]=vx[id+4*nnz];
                else                                    s_vx[threadIdx.x][threadIdx.y
                   +9]=0.0f;
        }
        __syncthreads();

        float diff2=1.1962890625000f*(s_vx[threadIdx.x][threadIdx.y+5]-s_vx[
           threadIdx.x][threadIdx.y+4])
                       -0.0897216796875000f*(s_vx[threadIdx.x][threadIdx.y+6]-
                          s_vx[threadIdx.x][threadIdx.y+3])+
                       0.0138427734375000f*(s_vx[threadIdx.x][threadIdx.y+7]-
                          s_vx[threadIdx.x][threadIdx.y+2])
                       -0.0017656598772321f*(s_vx[threadIdx.x][threadIdx.y+8]-
                          s_vx[threadIdx.x][threadIdx.y+1])+
                       0.0001186794704861f*(s_vx[threadIdx.x][threadIdx.y+9]-
                          s_vx[threadIdx.x][threadIdx.y]);
        convvx[ik]=bx[ik]*convvx[ik]+(bx[ik]-1.0f)*_dx*diff2;
        up[id]+=convvx[ik];
}

// update wavefield p
// if frsf==true, free surface boundary condition
__global__ void cuda_step_forward(float *vel, float *up, float *p0, float *p1,
    float dt, bool frsf, int npml, int nnz, int nnx)
{
        int i1=blockIdx.x*blockDim.x+threadIdx.x;
        int i2=blockIdx.y*blockDim.y+threadIdx.y;
        int id=i1+i2*nnz;
        float c=dt*vel[id];c=c*c;
        p0[id]=2*p1[id]-p0[id]+c*up[id];

        if (frsf && i1<npml)    p0[id]=0.0;
}


//=================================== read and write/save the boundary
    ==================================
// read and write the inner computation zone boundary coefficients from and into
     RAM along z direction
// read==flase, write/save boundary; read==true, read the boundary
__global__ void cuda_rw_boundarytb(float *boundarytb, float *p, int npml, int
    nnz, int nnx, int NJ, bool read)
{
        //int nx=nnx-2*npml;
        int nz=nnz-2*npml;
        int i1=threadIdx.x+blockDim.x*blockIdx.x;
        int i2=threadIdx.y+blockDim.y*blockIdx.y;
        int id=i1+2*(NJ-1)*i2;
        int i1p=i1+npml-(NJ-1);
        int i2p=i2;
        int idp=i1p+nnz*i2p;

        if (i1<NJ-1 && i2<nnx)
```

```
		{
			if(read)
			{
				p[idp]=boundarytb[id];
				p[idp+nz+NJ-1]=boundarytb[id+NJ-1];
			}
			else
			{
				boundarytb[id]=p[idp];
				boundarytb[id+NJ-1]=p[idp+nz+NJ-1];
			}
		}
}

// read and write the inner computation zone boundary coefficients from and into
    RAM along x direction
// read==flase, write and save boundary; read==true, read the boundary
__global__ void cuda_rw_boundarylr(float *boundarylr, float *p, int npml, int
    nnz, int nnx, int NJ, bool read)
{
	int nx=nnx-2*npml;
	//int nz=nnz-2*npml;
	int i1=threadIdx.x+blockDim.x*blockIdx.x;
	int i2=threadIdx.y+blockDim.y*blockIdx.y;
	int id=i1+nnz*i2;
	int i1p=i1;
	int i2p=i2+npml-(NJ-1);
	int idp=i1p+nnz*i2p;

	if (i1<nnz && i2<NJ-1)
	{
		if (read)
		{
			p[idp]=boundarylr[id];
			p[idp+nnz*(nx+NJ-1)]=boundarylr[id+nnz*(NJ-1)];
		}
		else
		{
			boundarylr[id]=p[idp];
			boundarylr[id+nnz*(NJ-1)]=p[idp+nnz*(nx+NJ-1)];
		}
	}
}



//======================================== imaging condition
    ======================================================
__global__ void cuda_cross_correlate(float *Isg, float *Iss, float *sp, float *
    gp, int npml, int nnz, int nnx)
{
	int i1=threadIdx.x+blockDim.x*blockIdx.x;
	int i2=threadIdx.y+blockDim.y*blockIdx.y;
	int id=i1+i2*nnz;

	if(i1>=npml && i1<nnz-npml && i2>=npml && i2<nnx-npml)
	{
		float ps=sp[id];
		float pg=gp[id];
		Isg[id] += ps*pg;
		Iss[id] += ps*ps;
```

```
        }
}


__global__ void cuda_imaging(float *Isg, float *Iss, float *I1, float *I2, int
    npml, int nnz, int nnx)
{
        int nz=nnz-2*npml;
        int nx=nnx-2*npml;
        int i1=threadIdx.x+blockDim.x*blockIdx.x;
        int i2=threadIdx.y+blockDim.y*blockIdx.y;
        int id=i1+i2*nnz;

        if(i1>=npml && i1<nnz-npml && i2>=npml && i2<nnx-npml)
        {
                I1[id]+=Isg[id];                // correlation imaging condition
                I2[id]+=Isg[id]/(Iss[id]+EPS);// normalized image
        }
        __syncthreads();

        if      (i2>=0  && i2<npml && i1>=0 && i1<npml)         // top left
        {       I1[id]=I1[nnz*npml+npml];               I2[id]=I2[nnz*npml+npml
            ];          }
        else if (i2>=npml+nx && i2<nnx && i1>=0 && i1<npml)     // top right
        {       I1[id]=I1[nnz*(npml+nx-1)+npml];        I2[id]=I2[nnz*(npml+nx
            -1)+npml];          }
        else if (i2>=0 && i2<npml && i1>=npml+nz && i1<nnz)     // bottom left
        {       I1[id]=I1[nnz*npml+(npml+nz-1)];        I2[id]=I2[nnz*npml+(npml
            +nz-1)];            }
        else if (i2>=npml+nx && i2<nnx && i1>=npml+nz && i1<nnz)// bottom right
        {       I1[id]=I1[nnz*(npml+nx-1)+(npml+nz-1)]; I2[id]=I2[nnz*(npml+nx
            -1)+(npml+nz-1)]; }
        else if (i2>=npml && i2<npml+nx && i1>=0 && i1<npml)    // top
        {       I1[id]=I1[nnz*i2+npml];                 I2[id]=I2[nnz*i2+npml];
                                }
        else if (i2>=npml && i2<npml+nx && i1>=npml+nz && i1<nnz)// bottom
        {       I1[id]=I1[nnz*i2+(npml+nz-1)];          I2[id]=I2[nnz*i2+(npml+
            nz-1)];             }
        else if (i2>=0 && i2<npml && i1>=npml && i1<npml+nz)    // left
        {       I1[id]=I1[nnz*npml+i1];                 I2[id]=I2[nnz*npml+i1];
                                }
        else if (i2>=npml+nx && i2<nnx && i1>=npml && i1<npml+nz)// right
        {       I1[id]=I1[nnz*(npml+nx-1)+i1];          I2[id]=I2[nnz*(npml+nx
            -1)+i1];            }
}

__global__ void cuda_laplace_filter(float *Img, float *laplace, float _dz, float
     _dx, int npml, int nnz, int nnx)
{
        int i1=threadIdx.x+blockDim.x*blockIdx.x;
        int i2=threadIdx.y+blockDim.y*blockIdx.y;
        int id=i1+i2*nnz;
        float diff1=0.0f;
        float diff2=0.0f;
        if(i1>=npml && i1<nnz-npml && i2>=npml && i2<nnx-npml)
        {
                diff1=Img[id+1]-2.0*Img[id]+Img[id-1];
                diff2=Img[id+nnz]-2.0*Img[id]+Img[id-nnz];
        }
        laplace[id]=_dz*_dz*diff1+_dx*_dx*diff2;
}
```

# 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.