Computational optimization of Elastic Reverse Time Migration

Gustavo Alves

ABSTRACT

The elastic forward and adjoint operators involve the computation and storage of many wave fields and models. I present two solutions to speed up the computation time of elastic operators. The first one uses multithread parallelization, while the second introduces Intel's Single Program Multiple Data Program Compiler (ISPC). Each solution shows a speed-up of about three times when compared to the original algorithm. I combine these improvements with better memory usage procedures that allow the processing of larger data sets. I show results of an Elastic Reverse Time Migration (ERTM) applied to the 2D synthetic Marmousi2 data set.

INTRODUCTION

Elastic imaging is an important topic in seismic exploration and seismology, due to the additional information about the subsurface contained in shear and surface waves modes. In seismic exploration, shear data can be used to better estimate angle versus offset (AVO) effects (Innanen, 2014), as well as estimate shear velocities in near surface. Shear velocities can help identify gas hydrate occurrences, which are a source of drilling hazards (Lu and McMechan, 2004; Heggland, 2004). In global seismology, mode conversions between compressional and shear waves serve to detect and characterize interfaces in the upper and lower mantle. A mathematical description of the adjoint method for elastic data in seismology can be found in Brytik et al. (2013).

Over the last few decades, there have been many methodologies developed to extend the concept of Reverse Time Migration (RTM) to the elastic case (Chang and McMechan, 1987, 1994; Denli et al., 2008; Yan and Sava, 2008). However, most of these methods attempt to apply the same scalar approach of acoustic based RTM to the elastic experiment (Yan and Sava, 2008). Therefore, their results contain artifacts that need to be filtered out from the final image. Such artifacts can be avoided by correctly modifying the imaging condition to include the vectorial nature of elastic data. This modified imaging condition is described in Alves (2015).

A caveat of the vectorial approach is an increase in memory required for storing wave fields of multi-component data. The extended number of fields used in the computation can make realistic 2D and 3D problems infeasible in current computer

architectures, leading to solutions that exchange memory requirements for added processing time.

I present two solutions that address the processing time involved in elastic wave propagation. The goal is to reduce additional processing time created by avoiding extensive memory use. This allows the computation of more realistically sized models in an acceptable processing time.

The first solution I propose makes better use of multithreading capabilities of current computer architecture by changing the stencils derived for adjoint operators in spatial derivatives.

Second, I introduce the use of Intel's SPMD Program Compiler, which allows for a more aggressive vectorization of operators. This makes computation of derivatives faster, leading to speed ups in processing time.

I apply these improvements to the 2D elastic Marmousi model (Martin et al., 2002). Marmousi2 is a multiparameter extension of the 1988 Marmousi model (Martin et al., 2002) and includes models for density, compressional and shear velocities. It is a 17 km wide by 3.5 km deep 2D model. It features hydrocarbon reservoirs with oil and gas anomalies, targeted at studying angle versus offset (AVO) in simple and complex geometries. I simulate one hundred shots along the surface, with a split-spread receiver configuration and 4 km maximum offset.

METHODOLOGY

Optimizations in memory usage

The changes in memory requirements I introduce in my elastic algorithm are standard practice in data processing. I make extensive use of **pointers** to avoid memory copies between arrays. I also apply several linear interpolation operators which subsample the arrays in time and space. These changes decrease the memory needed to allocate wave fields, while increasing the total processing time of the algorithm. In the next two sections, I describe the methods used to counter this increase.

Race conditions in Open MP

In order to calculate the adjoint of the forward modeling operator, I require the adjoints of individual spatial derivative operators. These can be obtained in a very straightforward way by considering that the adjoint of a stacking operator is a spreading operator. Claerbout (2010) shows an example of a forward and an adjoint coded in this way,

```
subroutine igrad1( adj, add,  xx,n,  yy   )
```

```
integer i,         adj, add,     n
real                            xx(n), yy(n)
call adjnull(      adj, add,  xx,n,  yy,n )
do i= 1, n-1 {
        if( adj == 0 )
                yy(i) = yy(i) + xx(i+1) - xx(i)
        else {
                xx(i+1) = xx(i+1) + yy(i)
                xx(i  ) = xx(i  ) - yy(i)
                }
        }
return; end
```

where **adj** $==$ **0** does a forward derivative (stacking), while **else** does an adjoint derivative (spreading).

While this formulation is easy to understand and implement, it is not easily parallelized. Consider the modified code below, that makes use of Open MP directives to take advantage of multicore processors,

```
subroutine igrad1( adj, add,  xx,n,  yy   )
integer i,         adj, add,     n
real                            xx(n), yy(n)
call adjnull(      adj, add,  xx,n,  yy,n )
!$OMP PARALLEL DO DEFAULT (SHARED) PRIVATE (i)
do i= 1, n-1 {
        if( adj == 0 )
                yy(i) = yy(i) + xx(i+1) - xx(i)
        else {
                xx(i+1) = xx(i+1) + yy(i)
                xx(i  ) = xx(i  ) - yy(i)
                }
        }
return; end
```

where **i** defines a private counter that is assigned individually to each thread. This implementation contains a race condition in its adjoint, where two concurring threads will try to update the same vector positions. In the case of a 2D operator, this situation can be more complicated, as I show for a 10^{th} order stencil of a first order derivative,

```
  subroutine Bx(adj,add,model,data)
    logical,intent(in)                    :: adj,add
    real,dimension(nz,nx),intent(inout)   :: model,data
```

```
  if(adj) then
    if(.not.add) model = 0.
    !$OMP PARALLEL DO SHARED(model,data) PRIVATE(ix,iz)
    do ix=1,nx
      do iz=5,nz-5
        model(iz+5,ix) = model(iz+5,ix) + c1*data(iz,ix)
        model(iz+4,ix) = model(iz+4,ix) + c2*data(iz,ix)
        model(iz+3,ix) = model(iz+3,ix) + c3*data(iz,ix)
        model(iz+2,ix) = model(iz+2,ix) + c4*data(iz,ix)
        model(iz+1,ix) = model(iz+1,ix) + c5*data(iz,ix)
        model(iz  ,ix) = model(iz  ,ix) - c5*data(iz,ix)
        model(iz-1,ix) = model(iz-1,ix) - c4*data(iz,ix)
        model(iz-2,ix) = model(iz-2,ix) - c3*data(iz,ix)
        model(iz-3,ix) = model(iz-3,ix) - c2*data(iz,ix)
        model(iz-4,ix) = model(iz-4,ix) - c1*data(iz,ix)
      enddo
    enddo
  !$OMP END PARALLEL DO
  else
    if(.not.add) data = 0.
  !$OMP PARALLEL DO SHARED(model,data) PRIVATE(ix,iz)
    do ix=1,nx
      do iz=5,nz-5
        data(iz,ix) = data(iz,ix) + (c1*(model(iz+5,ix) - model(iz-4,ix))+&
                                     c2*(model(iz+4,ix) - model(iz-3,ix))+&
                                     c3*(model(iz+3,ix) - model(iz-2,ix))+&
                                     c4*(model(iz+2,ix) - model(iz-1,ix))+&
                                     c5*(model(iz+1,ix) - model(iz  ,ix)))
      enddo
    enddo
  !$OMP END PARALLEL DO
  endif
endsubroutine
```

where the private index **ix** again causes a race condition in the adjoint. A possible solution to this problem is,

```
subroutine Bx(adj,add,model,data)
  logical,intent(in)                    :: adj,add
  real,dimension(nz,nx),intent(inout)   :: model,data

  if(adj) then
    if(.not.add) model = 0.
    do iz=5,nz-5
      !$OMP PARALLEL DO SHARED(model,data,iz) PRIVATE(ix)
```

```
      do ix=1,nx
        model(iz+5,ix) = model(iz+5,ix) + c1*data(iz,ix)
        model(iz+4,ix) = model(iz+4,ix) + c2*data(iz,ix)
        model(iz+3,ix) = model(iz+3,ix) + c3*data(iz,ix)
        model(iz+2,ix) = model(iz+2,ix) + c4*data(iz,ix)
        model(iz+1,ix) = model(iz+1,ix) + c5*data(iz,ix)
        model(iz  ,ix) = model(iz  ,ix) - c5*data(iz,ix)
        model(iz-1,ix) = model(iz-1,ix) - c4*data(iz,ix)
        model(iz-2,ix) = model(iz-2,ix) - c3*data(iz,ix)
        model(iz-3,ix) = model(iz-3,ix) - c2*data(iz,ix)
        model(iz-4,ix) = model(iz-4,ix) - c1*data(iz,ix)
      enddo
      !$OMP END PARALLEL DO
    enddo
  else
    if(.not.add) data = 0.
  !$OMP PARALLEL DO SHARED(model,data) PRIVATE(ix,iz)
    do ix=1,nx
      do iz=5,nz-5
        data(iz,ix) = data(iz,ix) + (c1*(model(iz+5,ix) - model(iz-4,ix))+&
                                     c2*(model(iz+4,ix) - model(iz-3,ix))+&
                                     c3*(model(iz+3,ix) - model(iz-2,ix))+&
                                     c4*(model(iz+2,ix) - model(iz-1,ix))+&
                                     c5*(model(iz+1,ix) - model(iz  ,ix)))
      enddo
    enddo
  !$OMP END PARALLEL DO
  endif
endsubroutine
```

where the adjoint is parallelized only over the **ix** index. The first problem with this solution is that the index **iz** is now solved as a serial and not parallel operation. The second and more serious problem is that updates to **model** are non-sequential in memory, with an inner loop over the slow index (**ix**) instead of the fast index **iz**. Both situations result in a significant loss of performance.

To better understand the problem, I show the first order derivative operator in matrix form,

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix} = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix}, \tag{1}$$

which represents how the output vector $\mathbf{Y}$ is constructed from elements of vector $\mathbf{X}$. The $-\mathbf{1}$ term in the last row of the matrix is excluded by looping the operation only up to row index $\mathbf{n-1}$ of $\mathbf{Y}$.

Now, I show the adjoint operator matrix, which is the conjugate transpose of the forward operator matrix,

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix}. \tag{2}$$

The adjoint operator matrix is a lower triangular matrix with different boundary conditions than the forward one. Therefore, the first row element $-\mathbf{1}$ needs to be treated in a separate loop than the one used in the forward. On the other hand, now both forward and adjoint operators can be expressed as stacking operators,

```
export void Ax_c_(uniform int adj, uniform int add, uniform int nx,
                  uniform int nz, uniform int ix, uniform float model[],
                  uniform float data[]){

  uniform float c1=35.0/294912.0;
  uniform float c2=-405.0/229376.0;
  uniform float c3=567.0/40960.0;
  uniform float c4=-735.0/8192.0;
  uniform float c5=19845.0/16384.0;

  if (adj==1) {
    uniform int i=ix*nz;
    if (add==0) {
```

```
      for (uniform int iz=0;iz<nz;iz++){
        model[iz+i]=0.f;
      }
    }
    foreach(iz=5...(nz-5)) {
      model[iz+i] += c1*data[iz+i-4*nz]
   +c2*data[iz+i-3*nz]
   +c3*data[iz+i-2*nz]
   +c4*data[iz+i-  nz]
   +c5*data[iz+i     ]
   -c5*data[iz+i+  nz]
   -c4*data[iz+i+2*nz]
   -c3*data[iz+i+3*nz]
   -c2*data[iz+i+4*nz]
   -c1*data[iz+i+5*nz];
      }
  }
  else {
    uniform int i=ix*nz;
    if (add==0) {
      for (uniform int iz=0;iz<nz;iz++){
        data[iz+i]=0.f;
      }
    }
    foreach(iz=5...(nz-5)) {
      data[iz+i] += c1*(-model[iz+i-5*nz])+
c2*(-model[iz+i-4*nz])+
c3*(-model[iz+i-3*nz])+
c4*(-model[iz+i-2*nz])+
c5*(-model[iz+i-  nz])+
c5*( model[iz+i     ])+
c4*( model[iz+i+  nz])+
c3*( model[iz+i+2*nz])+
c2*( model[iz+i+3*nz])+
c1*( model[iz+i+4*nz]));
    }
  }
}
```

In this formulation, arrays are represented as continuous vectors, with the index **ix** taken out of the function and parallelized through OMP routines. Memory accesses are done sequentially for both forward and adjoint, avoiding performance losses like in the previous example. The code above uses a variant of **C** programming language that was developed for the Intel SPMD Program Compiler. I present a detailed description of this implementation in the next section.

The Intel SPMD Program Compiler

The Intel SPMD Program Compiler (ISPC) was developed to improve performance in problems defined as Single Program Multiple Data (SPMD), where the processor executes the same operation many times over a long sequence of values. It's goal is to provide performance boosts on orders of 3x to 6x, depending on computer architecture, without the need for extensive changes to existing code. Pharr and Mark (2012) give a thorough description of this compiler.

I implement derivative operators used in elastic forward and adjoint algorithms using ISPC. The last code example in the previous section shows an example of ISPC optimization. I also introduce a wrapper function that allows the interface between **Fortran** and **C**, which I do not show here for brevity.

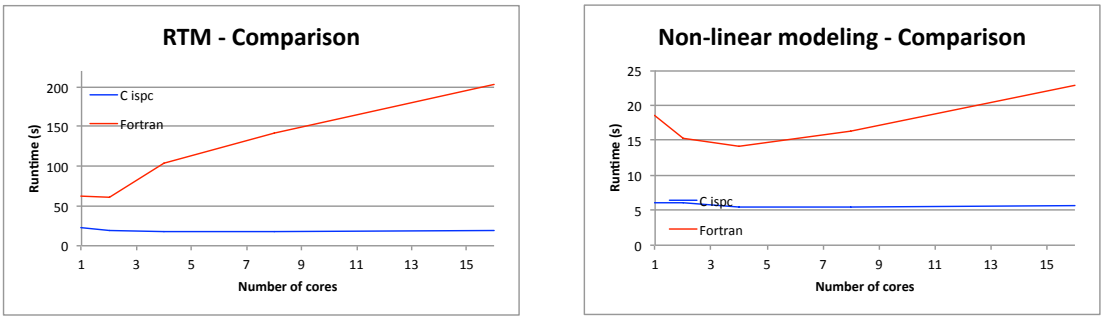
RESULTS

To test performance gains in the improved code, I ran an elastic non-linear modeling and migration for 5500 timesteps on a 300 by 300 2D grid and measured total time required by both forward and adjoint propagations. Figure 1(a) shows performance gains by re-writing the adjoint operators as stacking operators. Even for the best performance in the previous implementation, speed-up was about 3.4 times for 4 cores. The lowest time for ISPC (blue line) is also at 4 cores, which might be explained by the granularity of the computation. The linear increase of the original **Fortran** code as a function of number of cores shows performance decrease due to non-sequential data writes in swapped loops.

Figure 1(b) shows the improvement in computation time of the forward operator. Again, the best performance for both codes was at 4 cores. The ISPC implementation was about 3 times faster than the original code.

I apply the improved elastic algorithm to the the 2D Marmousi 2 model. First, I re-parametrize the model to density and Lamé parameters. I also change the sampling distance in space from 1.25 meters to 4 meters. Figure 2 shows the true model for density, λ and μ, respectively.

Even after subsampling, the complete model has 876 by 4251 grid points, which roughly corresponds to a 14.0 MB array. Forward modeling requires 30 model-sized arrays, which would correspond to 420.0 MB of allocated memory. However, migration requires 4045 model-sized arrays due to storing the background wave fields for crosscorrelation, considering a seismogram 1000 samples long. Total memory allocation in this case would correspond to 55.3 GB. Therefore, I divide the model domain into smaller 876 by 2000 grid points models and simulate 100 shots along the water surface, with a 4 km maximum offset split-spread configuration. This allows me to reduce the RTM memory cost to 33.6 GB per shot, as well as reduce the total number of computations required per shot. Receivers are placed at the same depth as the source, with 4 meter spacing. The source is an explosive Ricker type wavelet with a



(a) (b)

Figure 1: Computation time as a function of number of cores for the (a) RTM and (b) non-linear modeling algorithms. The red lines show the original codes and the blue lines the optimized ones. The optimal runtime for the **Fortran** implementation is at 2 and 4 cores for (a) and (b), respectively. For the ISPC implementation, optimal runtime is at 4 cores. [**NR**]

20 Hz peak frequency.

For each shot, I record seismograms for particle velocities and normal stresses. While the average of normal stresses corresponds to hydrophone data, which is usually recorded in marine acquisition, particle velocities are only recorded in ocean bottom surveys through the use of geophones. Therefore, a more realistic synthetic should include only hydrophone data or have receivers at the ocean bottom. After adding absorbing boundaries, each shot is computed in a 1036 by 2160 grid model. Five wave fields (particle velocities and stresses) are updated for 17017 time steps, totalling about 190.4 Giga samples computed. The optimizations I describe above achieve a performance of 105.25 Mega samples/second.

Figure 3 shows residual data for pressure, vertical and horizontal velocity. The shot position is $x = 8km$. I generate residual data by taking the data difference between an elastic non-linear modeling using the true (correct) model and and a background model. The background model I use has constant water properties and a positive gradient for all properties below the ocean floor, which makes the ocean-bottom primary weaker than in real data.

Figure 4 is the final elastic RTM image for density, bulk and shear moduli. I apply a vertical smoothing operator to remove low wavenumber noise close to shot positions and also a gain as a function of depth to improve imaging of deeper targets. The property models used in the migration are a smoothed version of the true ones, so the migrated image is very accurate.

These images lead to interesting observations regarding elastic RTM with this set of parameters. First, the density image shows stronger reflectors in deeper areas than bulk and shear. Second, there is crosstalk between images. Third, although the images for shear and bulk moduli seem similar, they have important differences.

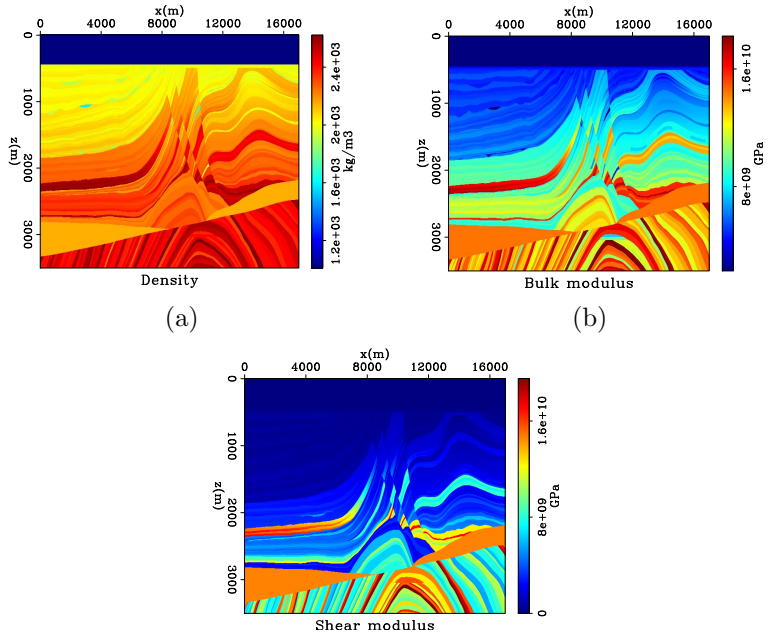


Figure 2: (a) Density, (b) λ and (c) μ for the Marmousi 2 elastic model. [**ER**]

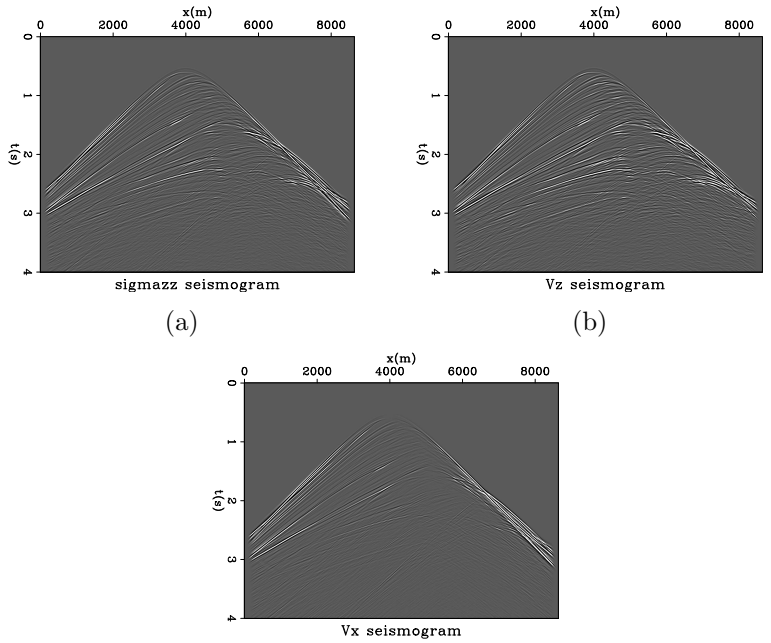


Figure 3: (a) Pressure, (b) vertical and (c) horizontal particle velocity for a sample shot at $x = 8km$. An absorbing boundary condition is applied at the sides for tapering the data. [**CR**]

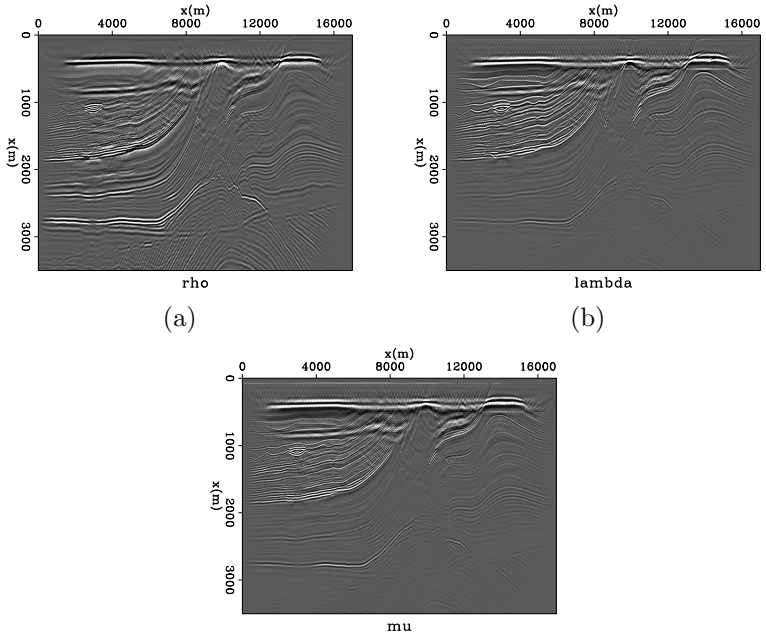


Figure 4: (a) Density, (b) bulk and (c) shear moduli images obtained after migration of 100 shots. Images were filtered with a vertical low cut to remove artifacts close to the sources and a gain was applied as a function of depth to better show deep reflectors. [**CR**]

Figure 5 shows the result of normalizing each image by its maximum amplitude and taking the ratio of the two. The resulting image is stronger where AVO anomalies are expected, i.e., when the vp/vs ratio changes drastically.

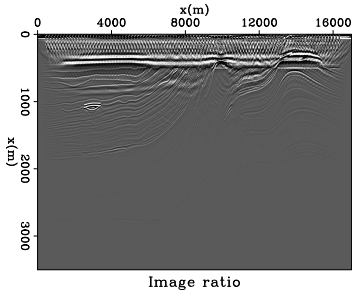


Figure 5: Image ratio between normalized bulk and shear moduli images. Notice how the reflectors with stronger vp/vs ratios get highlighted. [**CR**]

CONCLUSIONS

Computational optimization using Open MP routines resulted in a speed-up of over three times when compared to the original implementation. ISPC vectorization gave

an additional speed-up of three times, on par with expected values suggested by Intel.

Elastic RTM produces accurate images and can qualitatively show AVO anomalies. However, crosstalk between model parameters prevents a quantitative analysis. One possibility to address this issue is to apply an iterative inversion scheme, such as elastic full waveform inversion. Theoretically, such a scheme coupled with the correct preconditioning could remove crosstalk. This possibility requires further study.

ACKNOWLEDGEMENTS

I would like to thank SEP sponsors for the ongoing support and Petrobras for support of my PhD.

REFERENCES

Alves, G., 2015, Adjoint formulation for the elastic wave equation: Stanford Exploration Project Report, **158**, 133–150.

Brytik, V., M. V. De Hoop, and R. D. Van Der Hilst, 2013, Elastic-wave inverse scattering based on reverse time migration with active and passive source reflection data: Inverse Problems and Applications: Inside Out II, **60**, 411.

Chang, W.-F. and G. A. McMechan, 1987, Elastic reverse-time migration: Geophysics, **52**, 1365–1375.

———, 1994, 3-d elastic prestack, reverse-time depth migration: Geophysics, **59**, 597–609.

Claerbout, J. F., 2010, Basic earth imaging: Retrieved from http://sepwww.stanford.edu/sep/prof/bei11.2010.pdf.

Denli, H., L. Huang, et al., 2008, Elastic-wave reverse-time migration with a wavefield-separation imaging condition: Presented at the 2008 SEG Annual Meeting.

Heggland, R., 2004, Definition of geohazards in exploration 3-d seismic data using attributes and neural-network analysis: AAPG bulletin, **88**, 857–868.

Innanen, K., 2014, Reconciling seismic avo and precritical reflection fwi–issues in multiparameter gradient-based updating: Presented at the 76th EAGE Conference and Exhibition 2014.

Lu, S. and G. A. McMechan, 2004, Elastic impedance inversion of multichannel seismic data from unconsolidated sediments containing gas hydrate and free gas: Geophysics, **69**, 164–179.

Martin, G. S., S. Larsen, K. Marfurt, et al., 2002, Marmousi-2: an updated model for the investigation of avo in structurally complex areas: Presented at the 2002 SEG Annual Meeting.

Pharr, M. and W. R. Mark, 2012, ispc: A spmd compiler for high-performance cpu programming: Innovative Parallel Computing (InPar), 2012, 1–13.

Yan, J. and P. Sava, 2008, Isotropic angle-domain elastic reverse-time migration: Geophysics, **73**, S229–S239.