3D wave-packet decomposition implemented on GPUs

Victor V. Nikitin, Alexey A. Romanenko, Novosibirsk State University, Anton A. Duchkov, IPGG SB RAS and Novosibirsk State University, and Fredrik Andersson, Lund University*

SUMMARY

Decomposition of seismic data into wave-packet representations has been successfully used for 2D data compression, interpolation and de-noising. In this paper we present a fast implementation of a 3D wave-packet decomposition using graphical processing units (GPUs). This allows for the similar processing of 3D seismic gathers.

We discuss parallel implementation of the wave-packet transform on GPUs as opposed to existing algorithms (sequential and MPI-parallel). A few computational steps had to be modified adapting them for a GPU platform. The code has been tested on a 3D data set of size 256^3, where we obtain speedup of about 40 times compared to the sequential code performance.

INTRODUCTION

Wave packets provide a natural basis for the representation of seismic data and images (associated with singularities supported at traveltime curves or reflectors). They can be viewed as localized plane-waves; oscillatory in one direction (which can be associated with wavefront normal) and smoothly varying in the orthogonal directions (see Fig. 1). Thus, wave packets look suitable for 'building up' seismic waves. Different versions of wave-packet type basis functions can be found in the literature (Candés et al. (2006); Andersson et al. (2008); Fomel and Liu (2010)). It was also shown that wave-packet decomposition can be effectively used for data compression, regularization, interpolation, de-noising and imaging (Herrmann et al. (2007); Neelamani et al. (2008); Douma and de Hoop (2007)).

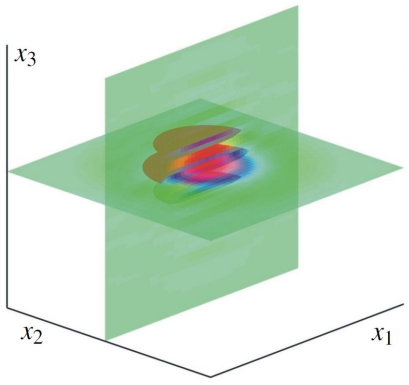

Figure 1: Example of a wave packet oriented vertically.

In this paper we consider a 3D wave-packet decomposition algorithm described in Duchkov et al. (2010). This implementation was ported on a graphical processing unit (GPU) platform using CUDA technology. Similarly to curvelet transforms our wave packet decomposition is implemented through Fourier transform. This is done using unequally spaced fast Fourier transforms (USFFT). The most computationally intensive part of USFFT is the weighted summation of values in a nearby grid points, *smearing*. This is an operation that is well suited for GPU implementation.

WAVE-PACKET DECOMPOSITION ALGORITHM

We briefly recapitulate the wave-packet decomposition algorithm described in details in Duchkov et al. (2010). We consider representations of 3D seismic data (or image) $u(\mathbf{x})$ of the form:

$$u(\mathbf{x}) = \sum_{\gamma} \tilde{u}_{\gamma} \varphi_{\gamma}(\mathbf{x}), \tag{1}$$

where $\tilde{u}_{\gamma}$ are wave-packet coefficients, and $\varphi_{\gamma}(\mathbf{x})$ are the wave packets. The wave packets $\varphi_{\gamma}(\mathbf{x})$ are chosen such that their Fourier transform have support in a box described by *scale* k (distance from the origin) and *orientation* ν. All boxes cover the 3D Fourier space (with coordinates $\boldsymbol{\xi}$) with some overlap as illustrated in Fig. 2.

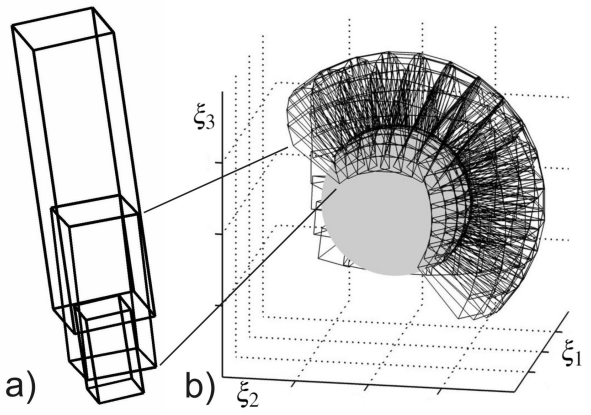

Figure 2: Box tiling of the Fourier space. (a) - overlapping in radial direction (different scales k), (b) overlapping in orthogonal direction (same scale but different orientations ν).

Let us assume that function $u(\mathbf{x})$ is sampled at equally spaced points, $\mathbf{x}_{\boldsymbol{i}}$, with multi-index $\boldsymbol{i} = (i_1, i_2, i_3)$, $i_k = -\frac{N}{2}, ..., \frac{N}{2}$. The regular grids $\boldsymbol{\xi}_{\boldsymbol{j}}^{\nu,k}$ are also defined on every box defined by (k, ν); $\boldsymbol{j} = (j_1, j_2, j_3)$, $j_k = 1, ..., N_k$. A forward wave-packet (WP) transform maps an input function into a set of coefficients:

$$\mathrm{F} : u(\mathbf{x}_{\boldsymbol{i}}) \to \tilde{u}_{(k,\nu,\boldsymbol{j})}, \tag{2}$$

where index $\boldsymbol{j}$ defines points on *translation grids* $\mathbf{x}_{\boldsymbol{j}}^{\nu,k}$ defin-

ing possible positions of central points of wave packets corresponding to scale k and orientation ν.

In order to get a discrete wave-packet transform of complexity comparable to FFT we make use of the USFFT as developed by Dutt and Rokhlin (1993) and Beylkin (1995). A discrete wave-packet transform along with adjoint (approximate inverse) can be schematically described as

$$u(\mathbf{x}_i) \xrightarrow{\text{USFFT}} \hat{u}(\boldsymbol{\xi}_l^{\nu,k}) \xrightarrow{\text{Windowing}} \hat{u}(\boldsymbol{\xi}_l^{\nu,k})\hat{\chi}_{\nu,k}(\boldsymbol{\xi}_l^{\nu,k}) \xrightarrow{\text{IFFT}} \tilde{u}_{(k,\nu,\boldsymbol{j})}, \quad (3)$$

and

$$u(\mathbf{x}_i) \xleftarrow{\text{AdjointUSFFT}} \hat{u}(\boldsymbol{\xi}_l^{\nu,k})\hat{\chi}^2_{\nu,k}(\boldsymbol{\xi}_l^{\nu,k}) \xleftarrow{\text{Windowing}} \hat{u}(\boldsymbol{\xi}_l^{\nu,k})\hat{\chi}_{\nu,k}(\boldsymbol{\xi}_l^{\nu,k}) \xleftarrow{\text{FFT}} \tilde{u}_{(k,\nu,\boldsymbol{j})}.$$

The last operation is a standard FFT transform on every box, and the second operation (windowing) is also just a multiplication of 3D arrays. The USFFT procedure is the most complicated as it requires interpolation of highly oscillatory spectrum from one grid to another with controlled accuracy. As proposed by Dutt and Rokhlin (1993) and Beylkin (1995), it consists of the following steps:

- input image preprocessing (doubling the grid adding zeros and multiplying by weighting coefficients);
- standard FFT operation on a regular grid;
- smearing operation.

The smearing operation is the final step of the interpolation and it takes about 90 % of total computational time necessary for the decomposition. Thus, we will discuss this operation in more detail in the following section while describing how to port the code to a GPU platform.

CODE MODIFICATION FOR GPU

Gathered smearing (forward transform)

The smearing step starts by having a spectrum of the input data on a regular (global) grid illustrated by the gray squares in Fig. 3,a. (Illustration is shown in 2D but the reader should imagine a 3D picture with boxes instead of rectangles.) Now, our aim is to finish the interpolation of the spectrum from the global grid onto rotated (local) grid on each box as shown by black crosses in Fig. 3,a. For each box we loop through local grids (black crosses), and for each cross we compute a weighted sum of the values at the gray squares located within the influence region depicted by red square with a side h_q in Fig. 3,a. We call this step *gathered smearing*. The parameter h_q is chosen as discussed in Dutt and Rokhlin (1993), and it is taken to be 6 in most of our applications.

It is natural to parallelize the calculation for each box by employing a number of threads equal to the number of points in the box grid (crosses). In order to perform the computation we have to load the following data into the GPU memory: a block of data on global grid containing the box; the corresponding window function; and the transform parameters.

After performing the smearing step for all the points on the box we can multiply it by a window function and use an inverse FFT for it getting a subset of wave packet coefficients. All these operations are executed locally on the GPU because all necessary data is already loaded in its global memory.

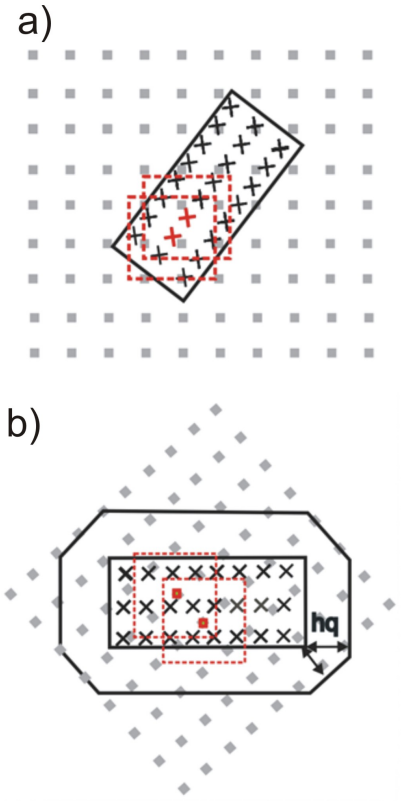

Figure 3: Layout of a global (regular) grid shown with gray squares and local (rotated) grid shown with black crosses.

Scattered smearing (inverse transform)

For the inverse transform we need to reverse the operation described in the previous section. We will call this reversed operation *scattered smearing*. In the original code this step was implemented in a straightforward manner and Fig. 3,a can be used to illustrate this step as well. A loop is performed through local grid points (crosses) which have prescribed values now. Thus instead of collecting the values from the surrounding global grid points, we do the opposite procedure – scatter values with some weights to the global grid points (gray squares) that are located within the attraction region shown by the red square. From the two overlapping red squares in Fig. 3,a one can see that parallelization of this loop will result in a memory writing conflict when several parallel threads are trying to access the same memory address for writing. Note that simultaneous reading from the same address is allowed without causing conflicts. Also, note that this was not an issue for the MPI-parallel implementation because each CPU had its own copy of the global grid (gray squares). These copies are merged during a

final global reduction communication step.

One possible solution could be to use atomic operations which are implemented in two stages: blocking access and then operation execution. We did not consider this option for our application as it will result in uncontrolled memory access delays. Instead we modified the smearing procedure as schematically illustrated in Fig. 3,b. Now, we organize a loop over global grid points (gray squares) in the block containing the box. To obtain the interpolated values in each point we may use the gathered smearing procedure described in the previous subsection. That is, to compute a weighted sum of contributions from different points of the local grid (black squares). After parallelization of this loop the local grid is accessed by different threads simultaneously for reading data but each point of the global is accessed by only one thread for writing.

From Fig. 3,b one can see that this alternative implementation results in some extra computations as there are more points in the global grid and for some of them the scattered smearing is collecting zeros (for example, for uppermost and lowermost gray squares located far away from the box). However, the extra computation cost is compensated by the avoidance of multi-thread memory writing conflicts.

OPTIMIZATION OF GPU CODE

For a better program performance it is important to organize the memory access in a proper manner. The GPU memory hierarchy includes register, shared memory, constant cache, texture cache and global memory. The register and shared memory sizes are small but it has an access time of only 4 clock cycles. The constant and texture cache are usually much larger and also have an access time of about 4 cycles, provided that there are no cache misses. Finally, the global memory access time is about 400-600 cycles. Thus, it is necessary to rely on constant and texture cache to speed up the program.

First, we note that there are some parameters frequently used during the decomposition procedure: number of scales; size and orientation of boxes; etc. We allocate a structure containing these parameters in the constant cache. Then, we notice that interpolation (smearing) procedure requires only reading from big three-dimensional blocks of data in global memory. Each thread requires about a thousand memory readings. Thus, performance can be significantly improved by binding a texture pointer to the corresponding memory blocks. Data access is becoming much faster using texture cache.

We use Compute Visual Profiler designed for C applications working on GPUs. It allows us to measure the execution time for the different parts of the code; thread size; the number of cache misses; and other parameters. A profiling of our original smearing implementation showed about 50 % of cache misses. In order to use cache optimally we have to organize memory access in a more sequential manner. In particular, our smearing procedures contains three nested loops over the indexes (i, j, k) when running through a red square in Fig. 3 but in 3D. We have modified it in such a way that reading takes place from the texture cache address $(a+k, b+j, c+i)$, where a, b and c are some shifts. This resulted in reduction of cache misses to about 15 %.

Our code is extensively using the 3D Fourier transforms implemented in the specialized library CUFFT. In this library, 3D FFTs are organized as a combination of 2D transforms. This can result in different FFT performance in the case of rectangular data blocks. The final inverse FFT (IFFT) is applied separately to each box that has rectangular form with different number of grid points in different directions (for example, $38 \times 38 \times 76$). In our original implementation, the IFFT on the box was executed as 76 calls of FFT for 38×38 grid. Instead, a faster implementation of the same procedure is possible with 38 calls of FFT for 76×38 grid.

For gaining efficiency we also recast to single-precision computations. This required to identify all variable types explicitly and to use specific single-precision functions: __expf(), __sinf(), __cosf() etc. Further optimization of the code will include proper choice of parameters, such as the number of thread blocks, the number of registers per thread, and the size of shared memory per block. We plan to do this work in future using CUDA Occupancy Calculator.

TESTING

We have tested the wave-packet implementation on different platforms. In addition to the sequential program executed on a single CPU we consider a parallel implementation based on MPI. We also test our modified algorithm on two GPU platforms: GeForce Quadro 4000 and NVidia Tesla C2050. The last GPU card developed in 2010 is based on the *Fermi* technology providing faster floating point operations and data access due to better hierarchical cache structure.

From the Table 1 one can see that using GPU platform one can gain speed-up about 45 times (forward transform) and 35 times (inverse transform) compared to sequential implementation.

platform	forw. (sec)	inv. (sec)
Tesla C2050 (GPU)	65.5	89.5
GeForce Quadro 4000 (GPU)	88.0	119.1
4xDual-Core AMD Opteron 2218 (MPI)	1123.8	1150.1
Intel core i7	2947.5	3137.0

Table 1: Performance for 256^3 image on different platforms.

Just to illustrate that our transform is consistent (approximate inverse property), we show the reconstruction of a 3D data cube. We tested our code on a synthetic 3D data cube of size 256^3: a union of 256 shot gathers for a 2D model (one reflector and a background velocity model with a low-velocity lens). One shot gather (two-dimensional slice through the cube) is shown in Fig. 4,a. After applying the forward wave-packet transform we keep only the N_{wp} largest coefficients and discard the remainder of the coefficients. Then we can apply the adjoint transform to obtain a reconstructed image. We measure the compression rate for representing seismic data in the

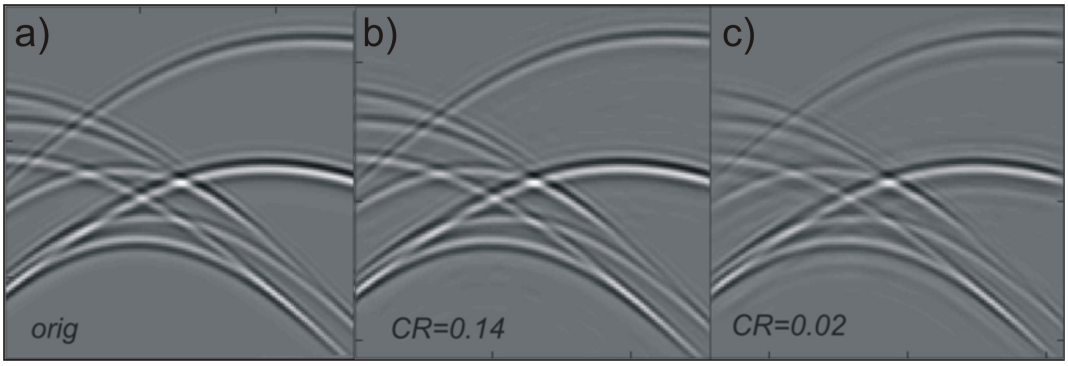

Figure 4: Wave-packet representation of 3D data cube. Slice through: (a) original data cube, (b) and (c) data reconstructed using different number of coefficients.

wave-packet domain by the ratio $CR = N_{wp}/N_{in}$, where N_{in} is the number of pixels in the original data cube.

Two-dimensional slices through the reconstructed cubes are shown in Fig. 4,b and Fig. 4,c for $CR = .14$ and $CR = .02$ correspondingly. One can see that there is no visible data quality deterioration in Fig. 4,b. In Fig. 4,c one can still see all relevant information about the wave kinematics although an aggressive thresholding results in a weakening of the amplitudes. This is not surprising if we look at the concentration property of the wave-packet coefficients. In Fig. 5 we show one particular box in the Fourier domain (gray sphere is centered at the coordinate origin). We draw only points of the local grid which correspond to large coefficients. They appear to be clustered in specific areas - where wavefield in data has orientation and frequency content defined in accordance with the chosen box.

DISCUSSION

For a 256^3 input function it is necessary to repeat smearing procedure $\sim 10^9$ times on a data array of 1 Gb size.

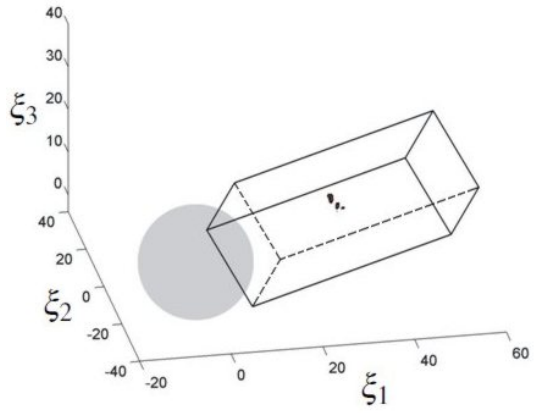

Figure 5: Large coefficients are shown for one box.

We note that it is possible to make further development in order to reduce memory requirements. First, we note that while working with real-valued functions only half of the the spectrum can be stored in memory. To deal with large images we propose to divide them into smaller segments processed separately (for example, 512^3 can be divided into 8 segments of size 256^3). Not that this is possible for seismic data because it is high-frequency in its nature. While dividing image into smaller segments we can loose information about the lowest frequencies. However, as the lower frequencies require a smaller covering in frequency range, we can deal with with them separately without increasing memory requirements.

Another important development is to adapt the code for decomposing rectangular data which usually has much more samples in time compared to number of receivers and sources. Dividing into segments may also help dealing with this problem.

We remind that the wave-packet decomposition algorithm was described in details in Duchkov et al. (2010), where we also discussed seismic applications of the code. Some successful applications of wave-packet type transforms were also reported in the literature (Herrmann et al. (2007); Neelamani et al. (2008); Douma and de Hoop (2007)).

CONCLUSIONS

An existing 3D wave-packet decomposition code was ported onto a GPU platform. This was done by modifying some of the computational steps of the algorithm. After some basic optimization we have got 45 and 35 times speedup for forward and inverse transform correspondingly. Speedup was measured against the sequential code for a data set of size 256^3.

This wave-packet decomposition code can be further used for 3D seismic data (or image) compression, interpolation and denoising.

ACKNOWLEDGMENTS

The work is partly supported by the Swedish Foundation for International Cooperation in Research and Higher Education and the Russian Ministry of education (GK P1178 from 03.06.10).

REFERENCES

Andersson, F., M. De Hoop, H. Smith, and G. Uhlmann, 2008, A multi-scale approach to hyperbolic evolution equations with limited smoothness: Communications in Partial Differential Equations, **33**, 988–1017.

Beylkin, G., 1995, On the fast Fourier transform of functions with singularities: Applied and Computational Harmonic Analysis, **2**, 363–381.

Candés, E., L. Demanet, D. Donoho, and L. Ying, 2006, Fast discrete curvelet transforms: SIAM Multiscale Model. Simul., **5 (3)**, 861–899.

Douma, H., and M. de Hoop, 2007, Leading-order seismic imaging using curvelets: Geophysics, **72**, S231–S248.

Duchkov, A., F. Andersson, and M. de Hoop, 2010, Discrete almost symmetric wave packets and multi-scale geometrical representation of (seismic) waves: IEEE Transactions on Geoscience and Remote Sensing, **48 (9)**, 3408–3423.

Dutt, A., and V. Rokhlin, 1993, Fast Fourier transforms for nonequispaced data: SIAM Journal on Scientific Computing, **14**, 1368–1393.

Fomel, S., and Y. Liu, 2010, Seislet transform and seislet frame: Geophysics, **75 (3)**, V25–V38.

Herrmann, F., G. Hennenfent, and P. Moghaddam, 2007, Seismic imaging and processing with curvelets: Extended Abstracts, EAGE 69th Annual Meeting.

Neelamani, R., A. Baumstein, and D. Gillard, 2008, Coherent and random noise attenuation using the curvelet transform: The Leading Edge, **27 (2)**, 240–248.