

# Linköping University Post Print

## Phase Based Volume Registration Using CUDA

Anders Eklund, Mats Andersson and Hans Knutsson

N.B.: When citing this work, cite the original article.

©2010 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Anders Eklund, Mats Andersson and Hans Knutsson, Phase Based Volume Registration Using CUDA, 2010, Proceedings of the 35th International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2010), 658-661.  
<http://dx.doi.org/10.1109/ICASSP.2010.5495134>

Postprint available at: Linköping University Electronic Press  
<http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-54035>

# PHASE BASED VOLUME REGISTRATION USING CUDA

*Anders Eklund, Mats Andersson, Hans Knutsson*

Division of Medical Informatics, Department of Biomedical Engineering  
Center for Medical Image Science and Visualization (CMIV)  
Linköping University, Sweden  
{andek, matsa, knutte}@imt.liu.se

## ABSTRACT

We present a method for fast phase based registration of volume data for medical applications. As the number of different modalities within medical imaging increases, it becomes more and more important with registration that works for a mixture of modalities. For these applications the phase based registration approach has proven to be superior. Today there seem to be two kinds of groups that work with medical image registration, one that works with refining of the registration algorithms and one that works with implementation of more simple algorithms on graphic cards for speeding up the algorithms. We put the work from these groups together and get the best from both worlds. We achieve a speedup of 10-30 compared to our CPU implementation, which makes fast phase based registration possible for large medical volumes.

*Index Terms*— Image registration, local phase, CUDA, GPU

## 1. INTRODUCTION

Image registration is needed in many cases and the goal is to transform a target image to a reference image such that the match is as good as possible. Common clinical application is to easier compare medical images from different modalities, such as MRI and CT, or to compensate for movement between or during scanning sessions. Since the obtained volumes can differ significantly in intensity, especially between different modalities, the local phase, from for example quadrature filters, is better to use since it is invariant to a change in intensity. Phase based image registration has for example been done by Hemmendorff et al. [1] and Mellor et al. [2] but none of them have implemented their algorithm on the graphics processing unit (GPU). GPU based image registration has been done by Bui et al. [3] and Ozelik et al. [4] but none of them use the phase based approach. Wong et al. [5] writes about fast phase based volume registration, but does not state any execution times in the article. Pauwels et al. [6] have implemented phase based optical flow on the GPU using Gabor filters. Their implementation is however for 2D and does not perform any registration, since the goal is to calculate motion vectors from video frames. They calculate a motion vector

for each pixel separately, while we solve an equation system that is setup globally, to achieve a parameter vector that best describes the transformation between the volumes. We then calculate a motion vector for each voxel from this parameter vector and achieve a movement field that varies smoothly. In this paper we take advantage of the phase based approach to image registration in 3D and use the parallel computing power of graphic cards at the same time.

### 1.1. Graphic cards and CUDA

CUDA, Compute Unified Device Architecture, is a parallel computing architecture developed by Nvidia. It enables the user to take advantage of the massive parallel computing power of the GPU. In CUDA, kernels (functions) are launched from the host (the CPU) but is executed on the device (the GPU). Each kernel is launched by a number of blocks, the grid, and a number of threads per block. In cases where we want to perform the same calculations for each pixel or voxel, the computational time can be reduced significantly since the graphics card performs calculations in parallel.

## 2. METHOD

### 2.1. Quadrature filters and local phase

A quadrature filter is a complex valued filter for combined edge and line detection. The real part of the filter, which is even, detects planes and the imaginary part, which is odd, detects 3D edges. The magnitude of the complex filter response tells us the phase invariant signal intensity and the phase tells us if there is an 3D edge or a plane. We use log-normal quadrature filters  $Q$ , which in the Fourier domain are expressed as spherical separable functions with radial function  $R$  and directional function  $D$  as function of frequency  $\mathbf{u}$ .

$$Q_k(\mathbf{u}) = R(\|\mathbf{u}\|)D_k(\mathbf{u}) \quad (1)$$

$$R(\|\mathbf{u}\|) = e^{C \ln^2\left(\frac{\|\mathbf{u}\|}{u_0}\right)} \quad C = \frac{-4}{B^2 \ln(2)} \quad (2)$$

Since the phase conception is valid only if we define a direction of it, we construct quadrature filters with different direction. The directions are defined such that

$$D_k(\mathbf{u}) = \begin{cases} (\mathbf{u}^T \hat{n}_k)^2 & \mathbf{u}^T \hat{n}_k > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where  $\hat{n}_k$  is the directional vector for filter  $k$ . The complex filter response  $q$  is an estimate of a bandpass filtered version of the analytical signal

$$q = A \cdot (\cos(\varphi) + i \cdot \sin(\varphi)) = A \cdot e^{i\varphi} \quad (4)$$

with magnitude  $A$  and phase  $\varphi$ . We use one filter in the  $x$ -direction, one in the  $y$ -direction and one in the  $z$ -direction. The filters we use have a centre frequency  $u_o = \frac{\pi}{3}$ , a bandwidth  $B = 1.7$  octaves and a spatial size of  $9 \times 9 \times 9$  voxels. To obtain filters with spatial locality and desired frequency response, advanced filter design is necessary [7].

## 2.2. Registration algorithm

Our registration algorithm is based on 2 assumptions

**I.** The motion can locally be described as a movement  $\Delta \mathbf{x}$ .

$$I(\mathbf{x}, t) = I(\mathbf{x} + \Delta \mathbf{x}, t + 1) \quad (5)$$

**II.** The image can locally be described as a leaning plane

$$I(\mathbf{x} + \mathbf{v}(\mathbf{x}), t + 1) = I(\mathbf{x}, t) + \nabla I^T \mathbf{v} - \Delta I \quad (6)$$

where  $\Delta I = I(\mathbf{x}, t) - I(\mathbf{x}, t + 1)$  and  $\nabla I = [\nabla_x I, \nabla_y I]^T$ .

The first assumption says that the intensity does not change between the two images. The second assumption says that the image locally can be described with a first order Taylor expansion. Combining these assumptions gives us the classical optical flow equation

$$\nabla I^T \mathbf{v} - \Delta I = 0 \quad (7)$$

The problem with the normal optical flow is that the assumptions that are needed are not met by many images. The phase of the filter response from quadrature filters is better suited for the assumptions, since it is invariant to a change in intensity and varies more smoothly. We then get the optical flow equation of the phase  $\varphi$

$$\nabla \varphi^T \mathbf{v} - \Delta \varphi = 0 \quad (8)$$

where

$$\Delta \varphi = \varphi_1 - \varphi_2 = \arg(q_1 q_2^*). \quad (9)$$

$q_1$  is the filter response from the target volume and  $q_2$  is the filter response from the reference volume and  $*$  denotes complex conjugation. The movement field is modelled with the help of a 12-dimensional parameter-vector according to

$$\mathbf{p} = [p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}]^T$$

$$\mathbf{v}(\mathbf{x}) = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + \begin{bmatrix} p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \\ p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \quad (10)$$

$$= \underbrace{\begin{bmatrix} 1 & 0 & 0 & x & y & z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & x & y & z & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & x & y & z \end{bmatrix}}_B \mathbf{p}$$

i.e. the first three parameters is the translation and the last 9 parameters make up the transformation matrix.

The phase gradients can be estimated with the following expression

$$\begin{bmatrix} \nabla_x \varphi \\ \nabla_y \varphi \\ \nabla_z \varphi \end{bmatrix} = \begin{bmatrix} \arg[q_{1x} + q_{1c}^* + q_{1c} q_{1x-}^* + q_{2x} + q_{2c}^* + q_{2c} q_{2x-}^*] \\ \arg[q_{1y} + q_{1c}^* + q_{1c} q_{1y-}^* + q_{2y} + q_{2c}^* + q_{2c} q_{2y-}^*] \\ \arg[q_{1z} + q_{1c}^* + q_{1c} q_{1z-}^* + q_{2z} + q_{2c}^* + q_{2c} q_{2z-}^*] \end{bmatrix} \quad (11)$$

where

$$\begin{aligned} q_c &= q(x, y, z) \\ q_{x+} &= q(x + 1, y, z), q_{x-} = q(x - 1, y, z) \\ q_{y+} &= q(x, y + 1, z), q_{y-} = q(x, y - 1, z) \\ q_{z+} &= q(x, y, z + 1), q_{z-} = q(x, y, z - 1) \end{aligned}$$

For each voxel and each filter, we also use a certainty that is calculated according to

$$c = \sqrt{|q_1 q_2|} \cos\left(\frac{\Delta \varphi}{2}\right)^2 \quad (12)$$

This certainty measure requires that we have a high magnitude both for the filter response from the reference volume and for the filter response from the target volume, and that the estimated phase does not differ too much.

If we use our model  $\mathbf{v}(\mathbf{x}) = B(\mathbf{x})\mathbf{p}$  and minimize the least square error

$$\epsilon^2 = \sum_k \sum_i c_{ik} (\nabla \varphi_k(\mathbf{x}_i)^T B(\mathbf{x}_i) \mathbf{p} - \Delta \varphi_k(\mathbf{x}_i))^2 \quad (13)$$

by setting  $\frac{\partial \epsilon^2}{\partial \mathbf{p}} = 0$ , we get the following equation system

$$\underbrace{\sum_k \sum_i c_{ik} B_i^T \nabla \varphi_{ik} \nabla \varphi_{ik}^T B_i}_{A} \mathbf{p} = \underbrace{\sum_k \sum_i c_{ik} B_i^T \nabla \varphi_{ik} \Delta \varphi_{ik}}_h \quad (14)$$

with the solution

$$\mathbf{p} = A^{-1} \mathbf{h} \quad (15)$$

Note that the equation system is easy to solve, 12 linear equations, but the cumbersome part is to sum over all voxels  $i$  and all filters  $k$ . By minimizing a  $L_2$  norm we can calculate the parameters that give the best solution. The most common approach is otherwise to maximize a similarity measure by searching for the best parameters, using some optimization algorithm. The complete algorithm uses the following steps in each iteration

- Filtering with 3 quadrature filters, that are complex valued in the spatial domain and not cartesian separable.
- Calculating phase differences, phase gradients and certainties for each filter and for each voxel, according to equations 9, 11 and 12 respectively.
- Setting up the equation system, i.e. calculating the A-matrix and the h-vector and solving the equation system to get the parameter vector.
- Calculating a motion vector for each voxel, according to equation 10, and using interpolation to get the value at that position from the modified volume.

To avoid the lowpass filtering effect from repeated interpolation in each iteration we accumulate the parameter vector in each iteration, i.e.  $\mathbf{p}_{total} = \mathbf{p}_{total} + A^{-1}\mathbf{h}$ , and always interpolate from the original target volume.

### 3. CUDA IMPLEMENTATION

#### 3.1. Filtering

The filtering was implemented as multiplication in the frequency domain. Filtering in the frequency domain is faster, than convolution in the spatial domain, if we have filters with many coefficients (729 complex valued in our case) and especially for filtering in 3D and 4D. The volume is first transformed with a 3D FFT, then the transformed volume is multiplied with the transformed filters and finally the filter responses are transformed with a 3D IFFT. We use a CUDA kernel for the complex multiplications between the filters and the volume. Since there is no fftshift function in the CUFFT library, in order to move the origin to the middle of the filter response volume from the corners, we instead perform a change of coordinate system.

#### 3.2. Phase differences, phase gradients and certainties

Since the calculation of the phase differences, phase gradients and certainties are exactly the same for each voxel, it is ideal for parallel computing on graphic cards. We use one CUDA kernel for calculating the phase differences and certainties and one kernel for calculating the phase gradients.

#### 3.3. Setting up the equation system

The A-matrix in the equation system is a 12 x 12 matrix and the h-vector is a 12 x 1 vector. There are, however, only 30 of the 144 elements in the A-matrix that need to be calculated, since the rest is zero or can be obtained from the fact that the A-matrix is symmetric, i.e.  $A(i, j) = A(j, i)$ . To setup the equation system in the CPU implementation we simply sum over all filters, voxels and non zero parameters. In the GPU implementation we take advantage of the high number of simultaneous threads by summing at many positions at the same time. The first CUDA kernel calculates the A-matrix

and h-vector for each parameter and at the same time sums over x, for each position (y,z). The second CUDA kernel sums over y, for each z, and the third CUDA kernel sums over z and creates the final A-matrix and h-vector.

#### 3.4. Solving the equation system

Since the equation system is very small, we send the A-matrix and the h-vector to the host and solve the equation system there. Only 12 floats need to be sent for the h-vector and 30 floats for the A-matrix.

#### 3.5. Motion vectors and interpolation

A big advantage with graphic cards, compared to CPU's, is that they have hardware support for interpolation, at least for linear interpolation, but cubic interpolation is also very fast. We use a 3D texture for fast interpolation and use a CUDA kernel that first calculates the motion vector, from the voxel's position (x,y,z) and the parameter vector  $\mathbf{p}$ , and then simply reads the value from the target volume at that position using a texture call.

## 4. RESULTS

The CPU implementation ran on an Intel Core 2 Quad with 4 cores at 2.83 GHz and 4 GB of memory. Our implementation ran on one of the four processor cores but all the execution times for the CPU implementation have been divided by 4 to make the comparison fair. The graphics card used was a Nvidia GTX 285 with 240 stream processors and 1 GB of memory. Neither the processor or the graphics card was over-clocked. As test volumes we used one synthetic test volume with a 3D cross with the resolution 128 x 128 x 128 voxels, and a MRI volume with the resolution 240 x 240 x 140 voxels, to which we applied translations and rotations. In the following sections we will present the computational times for the different parts in the algorithm, for the CPU implementation and for the GPU implementation, and the resulting speedups. We will also present the total time for 10 iterations of the algorithm. 10 iterations, on the original scale, is sufficient to for example compensate for 1-5 pixels of translation and 1-5 degrees of rotation in each dimension.

#### 4.1. Filtering

Volume size	CPU	GPU	Speedup
128 x 128 x 128	0.079 s	6.8 ms	11.6
240 x 240 x 140	0.31 s	187.7 ms	1.7

#### 4.2. Phase differences, certainties and phase gradients

Volume size	CPU	GPU	Speedup
128 x 128 x 128	0.60 s	7.8 ms	76.9
240 x 240 x 140	1.61 s	25.7 ms	62.6

### 4.3. Setting up the equation system

Volume size	CPU	GPU	Speedup
128 x 128 x 128	0.1 s	10 ms	10
240 x 240 x 140	0.33 s	50.2 ms	6.6

### 4.4. Solving the equation system

On average it takes 1 ms to send the data to the host and to solve the equation system there.

### 4.5. Motion vectors and interpolation

Volume size	CPU	GPU	Speedup
128 x 128 x 128	0.17 s	0.40 ms	425
240 x 240 x 140	0.30 s	1.44 ms	208.3

### 4.6. The whole algorithm, 10 iterations, including transfers to and from graphics card

Volume size	CPU	GPU	Speedup
128 x 128 x 128	9.48 s	0.31 s	30.6
240 x 240 x 140	29.80 s	2.97 s	10

### 4.7. fMRI-volumes

We also tested the implementation on a set of fMRI-volumes. In fMRI, functional magnetic resonance imaging, for example one volume per second is acquired to estimate a level of brain activity in each voxel. Since the subject can move the head during the acquisition, image registration is important. Each volume in the dataset had a resolution of 80 x 80 x 20 voxels and the dataset contained 324 volumes. 3 iterations per volume was sufficient and the registration took 5.1 s, or 0.0157 s per volume, compared to 0.1 s for a volume of size 64 x 64 x 30, for the fastest algorithm in the comparison by Oakes et al. [8]. This algorithm does not use any filters at all.

## 5. DISCUSSION

We have presented a method for fast phase based volume registration. Our implementation shows a speedup between 10 - 30, compared to our CPU implementation, for 10 iterations of the whole algorithm.

The filtering is significantly faster on the GPU. The problem with filtering in the frequency domain is however the limited memory size on the graphics card, since the filters require to be stored as volumes that are as big as the reference volume and target volume. For the test volume of size 128 x 128 x 128 we used the fast FFT developed by Nukada et al. [9]. This implementation has about 6 times better performance than the standard CUFFT library. The implementation however only works for the specific size 128 x 128 x 128 voxels, this is why the speedup for the filtering for the volume of size 240 x 240 x 140 is not as big since we have to use CUFFT in this case.

Calculation of the phase differences, phase gradients and certainties is very fast since it suits perfectly for parallelization. The setup of the equation system is significantly faster on the GPU since we can sum at many positions at the same

time. As expected, there is a huge performance gain by using the hardware support for trilinear interpolation when updating the compensated volume in each iteration. In the future we would like to use several scales such that large translations and rotations between the volumes can be handled by using a coarse scale first and then finer scales in the end. This would also significantly reduce the computational load even further.

## 6. ACKNOWLEDGEMENT

This work was supported by the Strategic Research Center MOVIII, funded by the Swedish Foundation for Strategic Research, SSF. We thank Akira Nukada et al. for letting us use their fast 3D FFT for volumes of size 128 x 128 x 128.

## 7. REFERENCES

- [1] M. Hemmendorff, M. Andersson, T. Kronander, and H. Knutsson, "Phase-based multidimensional volume registration," *IEEE Transactions on Medical Imaging*, vol. 21, pp. 1536–1543, 2002.
- [2] M. Mellor and M. Brady, "Non-rigid multimodal image registration using local phase," in *Lecture notes in computer science, Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 2004, vol. 3216, pp. 789–796.
- [3] P. Bui and J. Brockman, "Performance analysis of accelerated image registration using GPGPU," in *Proceedings of 2nd workshop on general purpose processing on graphics processing units*, 2009, pp. 38–45.
- [4] P. Muyan-Ozcelik, John D. Owens, Junyi Xia, and Sanjiv S. Samant, "Fast deformable registration on the GPU: A CUDA implementation of demons.," in *Proceedings of the 2008 International Conference on Computational Science and Its Applications (ICCSA)*, 2008.
- [5] A. Wong and P. Fieguth, "Fast phase-based registration of multimodal data," *Signal Processing*, vol. 89, pp. 724 – 737, 2008.
- [6] K. Pauwels and M. M. Van Hulle, "Realtime phase-based optical flow on the GPU," in *Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–8.
- [7] H. Knutsson, M. Andersson, and J. Wiklund, "Advanced filter design," in *Scandinavian conference on image analysis (SCIA)*, 1999.
- [8] T.R. Oakes, T. Johnstone, K.S. Ores Walsh, L.L. Greischar, A.L. Alexander, A.S. Fox, and R.J. Davidson, "Comparison of fMRI motion correction software tools," *Neuroimage*, vol. 28, pp. 529–543, 2005.
- [9] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.