Efficient 3D Isotropic Volume Reconstruction Based On 2D Localized Ultrasound Images

Jean-Baptiste Keck, *Student, Ensimag*
Matthieu Chabanas, *Supervisor, TIMC-IMAG Laboratory*

TIMC-IMAG

Abstract—A miniature 3D tracked ultrasonic probe has been developped to aquire intra-articular cartilage images under artroscopic chirurgy conditions. The aim is to detect cartilaginous lesions (arthrosis) and quantize their precise sizes and locations to help the clinician in his diagnostic and his therapeutic decision making. The ultrasonic transducer is tracked by an optical sensor, wich permits to find location and orientation of each 2D US images in a common 3D spacial reference. Near two thousands images are acquired when scanning a cartilage surface. An interesting tool is to rebuild a 3D isotropic volume (cubic voxels) with those images, allowing further processing. Conventional 3D ultrasound algorithms have low computational complexity but the huge amount of data generated makes it difficult to compute results within reasonable time on classical computers. In this paper we investigate the possibilities of regenerating a 3D isotropic volume with the help of GPGPU (CUDA) by adaptating existing algorithms to massive parallelism provided by modern everyday GPUs.

Index Terms—Introduction to Lab Research, Arthrosis, General Purpose Computing on Graphics Processing Unit, Ultrasound Imaging, Isotropic Volume Reconstruction.

1 Introduction

FREEHAND three-dimentional ultrasound imaging is a highly attractive research area because it is capable of volumetric visualization and analysis of tissue and organs. Conventional two-dimentional ultrasound imaging has been widely used for many clinical applications such as medical diagnosis and image-guided surgery. In comparisson with Computed Tomography (CT) and Magnetic Resonance Imaging (MRI), ultrasound imaging is non invasive, real time, portable and low cost. However, 2D ultrasound imaging fails to offer clinicians whole volume data for visualization and analysis. Thus three-dimensional ultrasound imaging systems has been developped to overcome those limitations by constructing various 3D datasets. Several approaches for constructing 3D ultrasound volume data have been reported. These approaches can be classified into three categories : dedicated 3D probes, mechanical scanning approach, and freehand scaning approach. Dedicated 3D probes can provide 3D data in real time but they are expensive and have limitations in scanning large volume organs. The mechanical scanning approaches usually use conventionals 2D transducers wich are translated and rotated with stepping motors. This design creates limitations in term of their scanning range. Finally, freehand ultrasound use conventionals 2D tranducers in pair with a positioning sensor to save position and orientation of each acquired image. Freehand 3D ultrasound has received increasing attention for it's low-cost and flexibility, as it allows the user to manipulate and view the desired anatomical section freely. During scanning a sequence of US images are captured along with their positions and orientations, asynchronously. Those datas are then used to reconstruct a 3D volume by using various interpolation or approximation algorithms. The reconstruction algorithm plays a key role in the construction

- *M. Keck is a student at Ensimag, Grenoble, France.*
- *M. Chabanas is in the team Gestes Médico-Chirurgicaux Assistés par Ordinateur in the TIMC-IMAG Laboratory, University of Grenoble, France.*

of three-dimentional ultrasound volume data with higher image quality and faster reconstruction speed.

TODO

Even if conventional 3D reconstruction algorithms have low computational complexity, the huge amount of data generated by a single scan (thousands of 2D images) makes it difficult to compute volume data in reasonable clinical time on classical computers. This paper aims to speed up 3D volume processing by adapting existing reconstruction algorithms to the massive parallelism provided by nowadays affordable GPUs. This is achieved throught General Purpose Computing on Graphics Processing Unit (GPGPU), a concept that has been well developped and widely adopted during the last decade and that continues its growth. The Open Computing Language (OpenCL) is the currently dominant open general-purpose GPU computing language. The dominant proprietary framework is the Compute Unified Device Architecture (CUDA). Althought CUDA requires a Nvidia CUDA compatible graphic card, we choose it because of its more mature status (performance and debugging tools) but algorithms are easily adaptable to OpenCL as well, allowing executions on Nvidia's and AMD's graphic cards, and even Integrated Graphics Processors (IGPs).

2 System overview

2.1 System setup

The freehand 3D ultrasound imaging system consits of three modules : a 2D ultrasound scanner specially designed to acquire intra-articular cartilage US images under artroscopic chirurgy conditions, an optical position and orientation sensor and a work station with custom-designed software for data-aquisition. The volume reconstruction is for the moment done as a post-process in another specially-designed piece of software. The portable ultrasound scanner concists of 64 axis-aligned transducers. Althought it is a freehand system, the axe is slightly rotated by a stepper motor to achieve a greater scanning area. The transformation due to the rotation of the motor is taken in account.

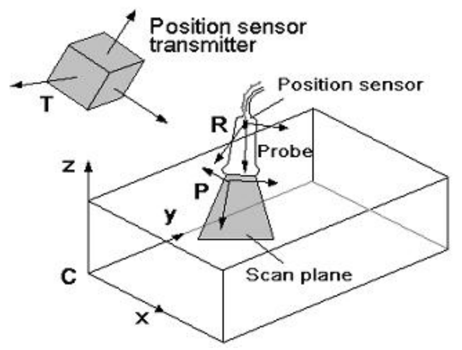


Fig. 1. A classical freehand ultrasound imaging system setup

The receiver of the spacial sensing device is attached to the hand-help probe of the ultrasound scanner. During data acquisition, spacial data and digitalized 2D scans are simulteanously recorded and collected. Since the ultrasound probe and the spacial sensor are independent, the temporal delay between two data streams can not be avoided.

2.2 Data acquisition

After signal processing 2D slices of logical size L_x x $L_y = 64$x1296 pixels are generated. As the transducers are $\epsilon_x = 205\mu m$ wide, the images have a physical size of $P_x = 64\ \epsilon_x = 19.12mm$. The precision on the other axe is $\epsilon_y = 18\mu m$, and thus the images have a physical height of $P_y = 1296\ \epsilon_y = 23.328mm$. For the images we will use this convention : we place the origin at the top-left with the x-axis oriented towards the right and the y-axis oriented towards the bottom. We define $X_p = (x_p, y_p, 0, 1)^T$ the homogenous vector that describes logical position of each pixel (x_p, y_p) in an image whose origin is at (O_x, O_y). S_x and S_y are the pixel spacing on each axis and are defined as $S_x = \frac{P_x}{L_x}$ and $S_y = \frac{P_y}{L_y}$.

$$M_{model} = \begin{pmatrix} S_x & 0 & 0 & O_x \\ 0 & S_y & 0 & O_y \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1)$$

The physical position of each pixel in its local physical coordinate system can be calculated as the following :

$$X_m = M_{model}\ X_p \tag{2}$$

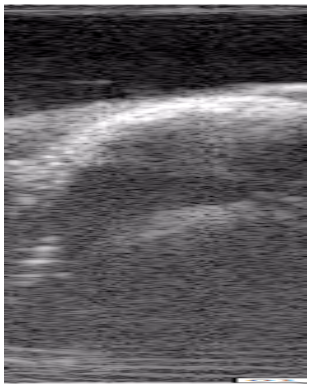


Fig. 2. A typical 2D ultrasound image of intra articular cartilage. The white band is the cartilage itself.

M_{RP} is the transformation matrix from the optical position sensor reflector (R) to origin of the scanned image plane (P), M_{ER} is the transformation matrix from the optical emitter (E) to (R) and M_{VE} is the transformation matrix from the voxel coordinate of the reconstructed volume (V) to (E). Note that all those matrices are 4x4 homogeneous matrices.

M_{RP} is initially unknown and must be obtained through a calibration process TODO.

With those matrices and (1), the whole forward transformation can be written as :

$$X_v = M_{VE}M_{ER}M_{RP}\ X_m = M\ M_{model}\ X_p \tag{3}$$

The specially designed sofware used to acquire data can export each image data in a raw file which includes single precision float image data, and a mhd file which include a path to the corresponding raw file and the transformation matrix M.

3 Volume Reconstruction

3.1 Reconstruction steps

After the 2D images and positions are exported as raw and mhd files, the next procedure is to reconstruct a 3D volume data. The sampled images are denoted $\{I_i\}$ and the transformations matrices $\{M_i\}$.

Here is the flow diagram of freehand 3D ultrasound volume reconstruction :

Step 1 : Data loading
- Parse mhd files
- Load raw images
- Load transformations

Step 2 : Data preprocessing
- Position filtering
- Image filtering
- Image cropping
- Image data conversion

Step 3 : Grid construction
- Determinate grid size

Step 4 : Volume Filling
- Bin-filling
- Hole-filling

3.2 Data loading

Loading data is not as easy as one might think and can rapidly become a serious bottleneck compared to what GPGPU can provide in terms of speedup. As each US scan is exported in its own mhd and raw file, thousands of mhd files have to be parsed to get each transformation matrix $\{M_i\}$ and get the path to the corresponding image data and load the raw image $\{I_i\}$. With two thousands images this operation can easily take 2 minutes on regular HDD[1] and even half a minute on a SSD[2]. One easy fix would be to merge all the image data into a single raw file, and the transformation matrices in another raw file as well, or simply keep the generated data in the computer memory when doing on the fly or immediate postprocessing. Data locality is something we do not only want on the disk, but in the memory too. This introduces an important concept : Array of Structures (AoS) versus Structure of Arrays (SoA). In GPGPU, memory accesses are a serious bottleneck when done improperly and SoA are often the better solution than AoS.

For example, a list of 3D vectors can be represented with 3 arrays, one for each of

1. Hard Disk Drive
2. Solid-State Drive

its coordinates (SoA), or with an array of 3D vector structures (AoS) :

Structure of Arrays (SoA):
```
struct vectors {
    float x[100];
    float y[100];
    float z[100];
}

struct vectors v;
```

Array of Structures (AoS):
```
struct vector {
    float x;
    float y;
    float z;
}

struct vector v[100]
```

When using a structure of arrays we have memory spacial locality between each of the vector coordinates. When a memory read occur in the global memory of a GPU, a large chunk of data is read (most of the time 256 bytes memory segments aligned on 256-byte adress). This is because the GPU is a massive parralel architecture. In CUDA, threads are grouped into thread blocks, which are assigned to multiprocessors on the device. During execution there is a finer grouping of threads into warps. Multiprocessors on the GPU execute instructions for each warp in Single Instruction, Multiple Data fashion. The warp size of current CUDA-capable GPUs is 32 threads. The device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize memory bandwidth. Any misaligned access by a half warp of threads results in 16 separate 32-byte transactions. If only 4 bytes are requested per 32-byte transaction (a simple integer or float), we should expect the effective bandwidth to be reduced by a factor of eight. For strided global memory access things get even worse, and get worser when the stride get bigger. That is not surprising because when concurrent threads simultaneously access memory addresses that are very far apart in physical memory, then there is no chance for the hardware to combine the accesses. Strides entailed by Array of Structures is the reason why we want to use Structure of Arrays, where data is nicely arranged to be accessed in a parallel manner.

Taking this into account, we decompose the 4x4 homogeneous transformation matrices $\{M_i\}$ into a structure of 12 arrays, the 3x3 inner transformation matrix coefficients r^1 through r^9, and the 3D offset vector coefficients (x, y, z) :

$$M_i = \begin{pmatrix} r_i^1 & r_i^2 & r_i^3 & x_i \\ r_i^4 & r_i^3 & r_i^5 & y_i \\ r_i^5 & r_i^6 & r_i^7 & z_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4)$$

The matrix is initially stored in a row-major manner, this is why we use such indexing convention for r_i.

The resulting struct is :
```
struct transformations {
    //offsets
    float *x;
    float *y;
    float *z;

    //3x3 matrix coefficients
    float *r1;
    float *r2;
    ...
    float *r9;
}
```

Image data do not recquire such memory reorganization as it is already stored and loaded as a row-major matrix of black and white pixels represented by simple precision floats. We just pay a special care to load the images in a big contiguous memory space and not to load them at random locations into the memory. Of course if the images were color images, we would have separated the red, green and blue channels into three separate arrays in a SoA manner.

This is not the only thing to take into account when loading data into memory when using GPGPU. When allocating CPU memory (RAM) that will be used to transfer data to the GPU memory (VRAM) for computing purpose, there are two types of memory to choose from : pinned and non-pinned memory. Pinned memory is memory specially allocated with a GPGPU dedicated malloc function (cudaMallocHost in CUDA), which prevents the memory from being swapped out and provides improved transfer speeds. Non-pinned memory is memory allocated using the classical malloc function in C, or using the new operator in C++. Pinned memory is much more expensive to allocate and deallocate but provides higher transfer throughput for large memory transfers. Empirically, using pinned memory only makes sense when the amount of memory transferred each way is larger than 16 MB (TODO). In our case with 2000 US scans, we have 664MB of float images, and 96Ko of transformations data so we just go for the pinned memory for the images.

At the end of this step, we have 12 arrays of transformation coefficients stored contiguously in the CPU memory in a non pinned manner, and one big contiguous array of floats representing the images that is pinned in the CPU memory.

3.3 Data preprocessing

Data preprocessing include image cropping, position filtering and image filtering.

Sometimes it is needed to crop the ultrasound images to select a region of interest. When doing this, we must not forget to keep or update image offset in local image coordinates for further processing.

Position filtering is needed because the position sensor device is susceptible to interferences and because our minitature probe was prone to bending. Thus the variation of probe position between consecutive slices can not be avoided. TODO

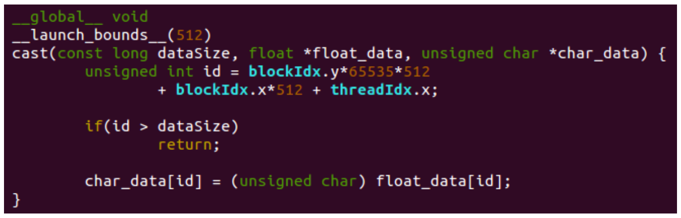
Volume reconstruction algorithms recquire accurate edge maps for good performance, howewer highly signal dependant nature of ultrasound speckle makes these difficult to obtain. Various filtering techniques have been developped to suppress speckles in order to improve the quality of images. Among them, the nonlinear filters have recently received an increasing interest, due to some of their important capabilities over linear filters. For example, A. Babakhani *et al.* [1] proposed in 2006 an extensive comparisson of such non linear filters applied to ultrasound imaging and 3D volume reconstruction, including non linear gaussian diffusion filters (NLG), anisotropic filters with level set (ANL), offset filters with level set (OFS) and pure morphological filters (PUM).

Due to a lack of time on this project, neither signal filtering nor image filtering have been considered here even if such filtering is possible on modern GPUs.

Most of current and past years GPUs have 512MB to 2GB embedded memory (VRAM). Thus a high volume of image data can be problematic when doing volume reconstruction on GPU because the grid needed to reconstruct the volume has to be stored too and memory should be kept for the GPU program execution stack. In addition to that, memory is already taken by the system through the graphic card driver when using a graphical desktop environment. One simple solution is to execute the algorithm on the GPU with smaller group of images each after another but transferring data from CPU memory to GPU memory is rather time-consuming. Typical transfer rates are 3Gbps for non-pinned memory and 6Gbps for pinned memory. We chose an even simpler method for reducing images impact on memory : For now our image were represented as an array of floats between 0.0 and 255.0 but we don't need such precision. Thus we decided to convert our 4 bytes floats into 1 byte unsigned chars, letting the initial 664MB images memory footprint fall down to only 166MB. This is done in a simple CUDA kernel, as it is a trivially parallelizable operation. To do this, we just have to allocate two arrays, a 166MB pinned memory array on the CPU and a 166MB memory array on the GPU, transfer initial float data to GPU memory, execute the kernel, transfer back the casted data to the CPU memory and finally free the old float datas on CPU and GPU.

Fig.3 shows a simple kernel performing this operation. Note the SIMD[3] coding style : there are simultaneous parallel computations, but only a single instruction at a given moment.

3. Single Instruction, Multiple Data



```
__global__ void
__launch_bounds__(512)
cast(const long dataSize, float *float_data, unsigned char *char_data) {
        unsigned int id = blockIdx.y*65535*512
                + blockIdx.x*512 + threadIdx.x;

        if(id > dataSize)
                return;

        char_data[id] = (unsigned char) float_data[id];
}
```

Fig. 3. A simple CUDA kernel example performing image data conversion from float to unsigned char.

Lets say N_i is the number of images we loaded at the first step. The $dataSize$ variable counts the total number of pixels ($dataSize = N_i$xL_xxL_y assuming we did not crop anything) and the id is a variable designating the current thread, starting at 0. Since the max id can exceed the number of pixels $dataSize$, a simple if statement prevent the program to access prohibited memory.

3.4 Grid construction

3.4.1 Computing bounding box

The third step in the reconstruction procedure is the establishment of coordinate system configuration for the reconstruction including its origin, its dimension and volume grid spacing. As our system has no need to predefine a volume before data acquisition we can use a simple bounding box technique as proposed by T. Wen *et al.* [2].

A bounding box can be represented with two points only, $X_{min} = (x_{min}, y_{min}, z_{min})^T$ and $X_{max} = (x_{max}, y_{max}, z_{max})^T$.

With the notations of Eq.2 and Eq.3, the algorithm to find the bounding box is pretty simple :

The test to know wether the point is in the bounding box or not concists of 6 scalar comparisons, and thus this algorithm can be executed safely on CPU without any performance drawbacks. The origin of the volume is then placed at X_{min}.

At the end of this algorithm, all US images are contained in this bounding box and X_{min} and X_{max} are atteigned by one ultrasound image corner at least once.

The next step is to choose a voxel size. This is a critical parameter because the memory footprint of the grid is what hinders GPGPU volume reconstruction.

```
**Data**: $X_{min} \leftarrow (+\infty, +\infty, +\infty)$
          $X_{max} \leftarrow (-\infty, -\infty, -\infty)$
**for** *each image* $I_i$ **do**
    **for** *each of the four corner vertice* $v_j$ *of* $I_i$ **do**
        $X_p^j \leftarrow$ image coordinate of v$_j$;
        $X_v^j \leftarrow M_i\ M_{model}\ X_p^j$;
        **if** *point* $X_v^j$ *outside of the bounding box* **then**
            Update $X_{min}$ and $X_{max}$;
        **end**
    **end**
**end**
```

Algorithm 1: Bounding box algorithm

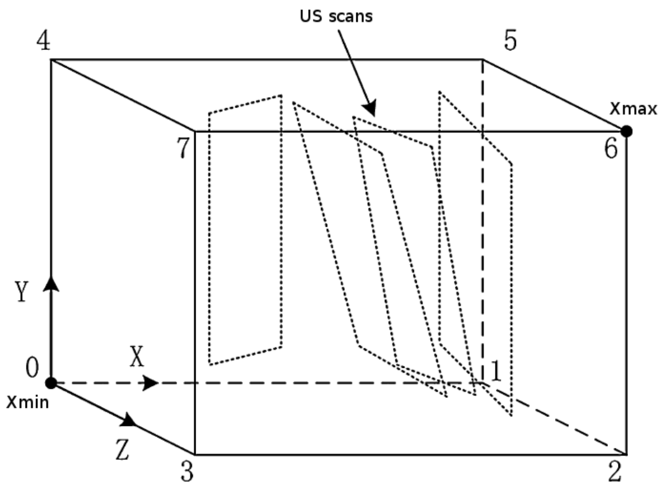


Fig. 4. The resulting bounding box defined by X_{min} and X_{max}.

3.4.2 Choosing a voxel size

Lets assume W_bxH_bxL_b are the width, height and length of the generated bounding box, W_gxH_gxL_g are the width, height and length of the grid (in voxels) and δ_v is the cubic voxel size. With those notations, we need to allocate at least a grid of $N_v = W_g$xH_gx$L_g = \frac{W_b}{\delta_v}x\frac{H_b}{\delta_v}x\frac{L_b}{\delta_v}$ voxels.

Thus, memory cost is cubical with the number of subdivisions N_s. Each time we want to double precision, we have to pay eight times more memory.

TABLE 1
Memory footprint for a 50x50x50mm bounding box and unsigned char voxels

$log_2(N_s)$	N_s	$\delta_v(\mu m)$	N_v	Grid memory
12	4096	12.2	2^{36}	68.7GB
11	2048	24.4	2^{33}	8.59GB
10	1024	48.8	2^{30}	1.07GB
9	512	97	2^{27}	134MB
8	256	195	2^{24}	16.8MB
7	128	390	2^{21}	2.1MB

Table 1 shows grid memory consumption for a bounding box which is approximately the size of the bounding box obtained when scanning our intra-articular cartilage. We see that when the voxel size δ_v goes under $0.1mm$ the memory begins to cause some problems to our GPU memory. Even if some professional graphic cards can have up to 12GB of memory, a simple δ_v set to $0.05mm$ can cause trouble. In fact, as it is explained later in this paper, even simple volume reconstruction algorithm will recquire at least four such grids in the bin-filling stage.

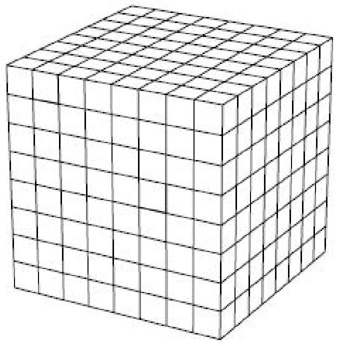


Fig. 5. A cubic bounding box split with $N_s = 8$

For example to perform an average in a Single Instruction, Multiple Data manner, we need at least two unsigned int grid and two unsigned char grids witch multiply the table 1 grid memory footprint by 10 when assuming unsigned int are 4 bytes and unsigned char 1 byte. That means that $\delta_v = 0.05mm$ entails at least $10.70GB$ memory consumption just for the grids on the GPU memory. Adding to that the size of the images, the system graphical environment load and the execution stack and our 12GB professional graphic card wont survive under such memory load.

The only solution is to split the grid into subgrids on the GPU side, and keep the whole voxel grid on the CPU side. The amount of splitting depends of the graphic card runtime available emory and the reconstruction algorithm used.

3.4.3 Splitting the grid

To ease grid splitting we round each grid side voxel size W_g, H_g, and L_g to upper power of two to get a grid surrounding our bounding box :

$$W_G = 2^{\lceil log_2(W_g) \rceil} \tag{5}$$
$$H_G = 2^{\lceil log_2(H_g) \rceil} \tag{6}$$
$$L_G = 2^{\lceil log_2(L_g) \rceil} \tag{7}$$

This is far from being an optimal splitting method as a simple layer of voxel in each directions can enlarge the grid by a factor of 8 and thus multiply by 8 memory footprint and computation time. If execution time is still critical after having ported volume reconstruction algorithms to GPGPU, adapting the code to handle different grid sizes is worth the investment.

One of the greatest benefit of having computed the upper power of two grid is that we can divide the grid in power of two subgrids and all the subgrids will have the same dimension, simplifying the implementation. Moreover this could be used for further optimisation when using multi-resolution grids to reduce global grid memory footprint.

Fig.6 shows the same grid as before. Because W_g, H_g and L_g were already powers of two, the surrouding power of two grid is the same. The grid is split with three parameters, S_x, S_y and S_z wich represent the splitting amount along the x, y and z-axis (width, height, and length). Those parameters are powers of two. The resulting subgrid is like the original grid, a power of two grid. The numbers of subgrids is given by the simple equation :

$$N_{SG} = S_x \, S_y \, S_z \tag{8}$$

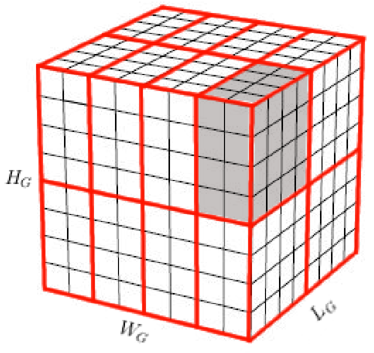


Fig. 6. The same cubic grid split in subgrids with $S_x = 4$, $S_y = 2$ and $S_z = 2$. One of the 16 subgrids is represented in grey.

The size of those subgrids is given by :

$$W_{SG} = \frac{W_G}{S_x} \quad (9)$$

$$H_{SG} = \frac{H_G}{S_y} \quad (10)$$

$$L_{SG} = \frac{L_G}{S_z} \quad (11)$$

```
Data: The size of the grid $W_G = 2^p$,
      $H_G = 2^q$ and $L_G = 2^r$
      The minimum upper power of two
      splitting ratio $S_R = 2^s$
Result: The splitting ratio on each axis
      $S_x = 2^i$, $S_y = 2^j$ and $S_z = 2^k$

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$;
while $s > 0$ do
    if $p>q$ & $p>r$ then
        $i \leftarrow i + 1$;
        $p \leftarrow p - 1$;
    else
        if $q>r$ then
            $j \leftarrow j + 1$;
            $q \leftarrow q - 1$;
        else
            $k \leftarrow k + 1$;
            $r \leftarrow r - 1$;
        end
    end
    s$\leftarrow s - 1$;
end
```

Algorithm 2: Splitting algorithm

3.4.4 Splitting algorithm

The grid is, like the images, stored in a contiguous array in the CPU memory. Accessing the voxel (i,j,k) is done by accessing the voxel at position kxW_Gx$H_G + j$x$W_G + i$ in the array. Because we want to keep data locality within the grid, we give splitting priority on the z-axis, the y-axis and finally the x-axis.

Once we have choosen a volume reconstruction algorithm, we know exactly how much space we will need for a given voxel size δ_v. After fetching GPU available memory, we can compute a minimum grid splitting ratio Sr. As we can only divide by a power of two, we take the upper power of two of this ratio, $S_R = 2^{\lceil log_2(S_r) \rceil}$.

After algorithm 2 we just need to allocate a grid of the size of one subgrid on the GPU and execute volume reconstruction algorithm on each of the subgrids one after another.

4 Volume Filling

Volume filling is the key procedure in the freehand 3D ultrasound systems. Various types of reconstruction algorithms have been reported and evaluated in TODO. These algorithms can be grouped into three categories : Voxel Nearest Neighbtor (VNN), Pixel Nearest Neightbor (PNN) and Distance Weighted (DW) interpolation algorithms. More elaborated algorithms methods are based on radial basis functions, such as spline interpolation functions TODO, or statistical Bayesian model with Rayleigh distribution TODO, but they are not suited for isotropic volume reconstructions (voxels).

4.1 Existing algorithms

PNN is the most intuitive method. It traverses each voxels, finds its nearest pixel by computing the shortest distance between the voxel and the sampled US images and inserts the nearest pixel value to the voxel TODO. Altought this algorithms can preserve the most original texture, ultrasonic echo with speckle noise, it also trends to generate large artifacts when the minimum distance to pixel becomes large.

VNN interpolation method is the most popular reconstruction algorithm, which traverses on each pixel in the US images and assigns the pixel value to the nearest voxel. The algorithm is done in two stages :

- **Bin-filling:** In the bin-filling stage each pixel is traversed and its pixelvlue is assigned to its nearest voxel. For a given voxel, multiple pixel contributions are generally handled by averaging pixel intensities.
- **Hole-filling:** In the hole-filling stage, the algorithm traverses on each voxel and fills each empty voxels by local neighborhood averaging. Most hole filling algorithms depend on the interpolation gaps, and there can still be few holes if the the distance among sampled US images is to far apart, greater than the interpolation radius.

With the VNN method, obvious artrifacts can observed on the boundaries between the highly detailed bin-filled regions and the smothed hole-filling regions.

Similar to the PNN interpolation method, DW interpolation proceeds voxel by voxel but instead of using the nearest pixel, each voxel value is assigned with a weighted average of pixel situated nearby. The parameters to choose are the weighting function, and the size and shape of the neighborhood. The simplest approach employs a spherical neighborhood. All the pixels in the sphere are weighted by the inverse distance to the voxel and are then averaged. If the radius is too large, the reconstructed volume will be highly smoothed.

For the interpolation stage (hole-filling), we can use the Fast Marching Method (FMM), proposed by T. Wen *et al.***??**. The proposed marching process ensures that the direction of information propagation is the normal direction of the evolving boundary, improving edge presevation in the hole-filled areas.

According to T. Wen *et al.***??**, average interpolation error in gray level for various datasets and for various amount of data removed, is the best for the FMM method, followed by the PNN method, the classical DW method and finally the VNN method.

5 Adapting volume filling algorithms to Cuda

6 Conclusion

The conclusion goes here.

Appendix A Voxel Renderer

Appendix text goes here.

Acknowledgments

I would like to thank the TIMC-IMAG laboratory for having hosted me for 4 months, especially the GMCAO team members that were always here to help when there were any problems and for the good mood. Special thanks goes to M. Chabanas for having taken the time and energy throughout the project by providing precious feedbacks and for having introduced me to the TIMC-IMAG laboratory.

[1] [2] [3] [4] [5] [6] [7].

References

[1] A. Babakhani, Z. Du, L. Sun, M. A. Fereidoon, and K. M. Reza, "3d reconstruction of ultrasonic images based on matlab/simulink," *Pakistan Journal of Biological Sciences*, vol. 9, no. 15, pp. 2818–2822, 2006.

[2] T. Wen, Q. Zhu, W. Qin, L. Li, F. Yang, Y. Xie, and J. Gu, "An accurate and effective fmm-based approach for freehand 3d ultrasound reconstruction," *Biomedical Signal Processing and Control*, vol. 8, no. 6, pp. 645–656, 2013.

[3] T. Qiu, T. Wen, W. Qin, J. Gu, and L. Wang, "Freehand 3d ultrasound reconstruction for image-guided surgery," *Bioelectronics and Bioinformatics (ISBB)*, pp. 147–150, 11 2011.

[4] M. Hafizah, T. Kok, and E. Supriyanto, "Development of 3d image reconstruction based on untracked 2d fetal phantom ultrasound images using vtk," *WSEAS Transactions on Signal Processing*, vol. 6, no. 4, pp. 145–154, 2010.

[5] J. H. R., H. G. Andrew, M. T. Graham, and W. P. Richard, "Sensorless reconstruction of unconstrained freehand 3d ultrasound data," *Ultrasound in Medicine and Biology*, vol. 33, no. 3, pp. 408–419, mar 2007.

[6] W. Huang and Y. Zheng, "Mmse reconstruction for 3d freehand ultrasound imaging," *International Journal of Biomedical Imaging*, no. 2, p. 8, jan 2008.

[7] H. Yu, M. S. Pattichis, C. Agurto, and M. B. Goens, "A 3d freehand ultrasound system for multi-view reconstructions from sparse 2d scanning planes," *BioMedical Engineering OnLine*, vol. 10, no. 7, p. 8, jan 2011.