PHOTONICS PUBLIC PRIVATE PARTNERSHIP

INNODERM

687866 —H2020-ICT-2015

D3.3: Report on algorithmic acceleration D3.4: Report on read electronics speed and efficiency

Deliverable nature:	*Report*
Contractual delivery date:	02.2019
Actual delivery date:	02.2019
Version:	1.0

Dissemination level (select one)		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Lead beneficiary for deliverable: RayFos

Author(s): Patrick O'Driscoll, Vassilis Sarantos, Mathias Schwarz

Organisation (s):
RAY – RayFos, ITH - iThera

E-mail: patrick.odriscoll@rayfos.com, vassilis.sarantos@rayfos.com, mathias.schwarz@ithera-medical.com
Phone: +44 2035192444, +49 89 7007449 72

Table of Contents

Purpose of this document

This document's aim is to report and document the progress made in enhancing, developing and producing software to accelerate the algorithm used by our GPU hardware creating an image of intensity values within a region of interest. We have described the events occurring along the timeline of development, along with explanations and examples of the code developed and decisions made so far. Graphs and charts give a picture of the performance enhancements and follow the flow of decisions made along the project.

Introduction

The acceleration of the algorithm utilised in this application calculates a reconstructed volume generated from a scanner run over a patient's skin. This is the typical raster scan of the RSOM system and is usually displayed as a set of maximum intensity projection images. The basis of this algorithm which already existed in an initial CUDA implementation was already used in the 1st generation RSOM (Raster Scan Optoacoustic Mesoscopy) from iThera. We initially ported this code to CUDA v8 (from a previous CUDA version) before starting a number of iterative attempts to improve performance. Although the code was already using GPU acceleration (Nvidia-CUDA) the processing time was still rather significant (a few minutes on fast GPUs), which is a prohibiting figure for any attempt to support multi-spectral processing, real-time preview and fast image results once the raster scan is complete. The computational complexity of the algorithm is high because processing requires calculating a huge number of voxels for a high number of "virtual" sensor positions that is equal to the number of the raster scan points. The voxels consume huge amounts of data, as is common in SAFT algorithm applications (see below for explanation); hence the computation performance is paramount and GPU acceleration an absolute requirement.

The generalised SAFT algorithm

In the SAFT model the focal point of the transducer is considered to be a virtual point detector. When tissues absorb pulsed laser energy, photoacoustic (PA) waves generated within a particular angle are considered detected by the virtual detector. The algorithm consists of applying appropriate delays relative from the virtual point detector to the adjacent scan lines and then summing all the delayed signals, producing a signal or intensity value for a particular point in time:

$$S_{SAFT}(t) = \sum_{i=0}^{N-1} S(i, t - \Delta t_i)$$

Here the S(i,t) is the received signal at scan line i and the Δt is the calculated acoustic propagation time from the synthetic focal point to the virtual detector at scan line i. The N is the maximum number of scan lines included in the sum, which is determined by the extent of the PA radiation. The negative time delay is used if the synthetic focal point is lower than the focal point of the transducer.

The SAFT algorithm has many applications, but in the ultrasonic imaging context its optimisation is of the utmost significance. After the code was initially handed over and ported to CUDA, it was being applied to calculate the intensity values using a so-called Z-slab approach.

The SAFT algorithm presented above was further generalised and modified with regard to the detector properties. The summation over the signal was modified to a sum over a weighting factor modelling the sensitivity field of the detector multiplied by a back-projection term. The back-projection term is a combination of the direct received signal and its derivative. More details on the generalised algorithm are presented in the dissertation (Schwarz, Mathias. Multispectral Optoacoustic Dermoscopy: Methods and Applications. Diss. Technische Universität München, 2017).

The Z-Slab approach

Z-Slab Workflow

The cube shown below in Figure 1 represents the entire volume of data that can be reconstructed by the raster scan movement of the transducer; because of the amount of voxel data that need to be calculated (typical count 500x500x500 voxels) it is not possible to process the whole volume in a single pass of the GPU accelerated function. In order to slice the problem in more manageable blocks, the Z-slab approach splits the volume in separate strips (slabs) along the depth of the cube (Z-axis). Each call to the GPU accelerated function, calculates a single slab by executing multiple processing threads in the GPU so as to calculate the voxel values of this slab. On each function call, the algorithm calculates only the voxels in the area shown highlighted in grey (shown below). In order to calculate the whole volume, the software needs to call multiple times the GPU accelerated function until it processes the whole volume (typical number of Z-Slabs ~20). It must be noted here that it is assumed that the dimensions of the virtual sensors grid in the SAFT algorithm described above are equal to the dimensions of the volume voxels in X and Y axis.

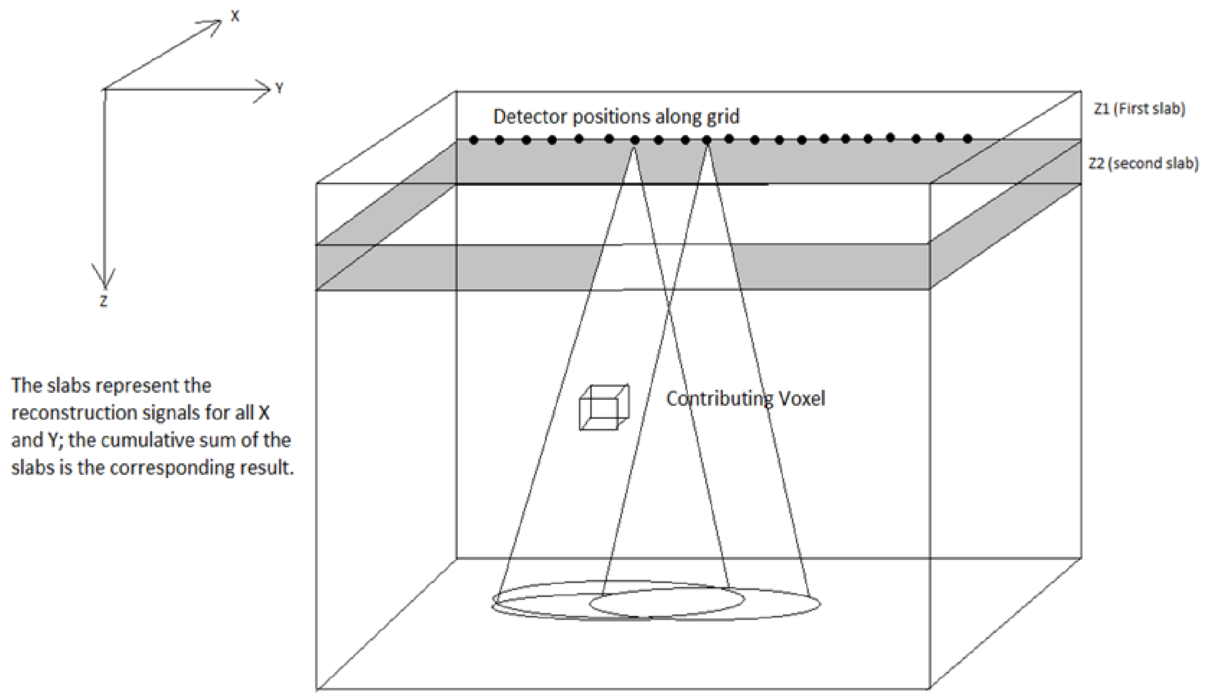
X
Y
Z
Detector positions along grid
Z1 (First slab)
Z2 (second slab)
Contributing Voxel
The slabs represent the reconstruction signals for all X and Y; the cumulative sum of the slabs is the corresponding result.

Figure 1: Z-slab approach of cube data volume

In order to be able to analyse and debug the original CUDA code we decided to first port the code to standard C that could be executed on a CPU. Although GPU development tools have evolved since the early days of GPU frameworks, it is still cumbersome to debug on OpenCL or CUDA platforms and it is a good practise to first implement alternate algorithmic approaches on C code running on CPUs before attempting to port to CUDA or OpenCL. Obviously, the standard C Code solving this problem while running on CPU, without parallelisation is dramatically slow; due to the structure of the SAFT algorithm and the amount of data and calculations involved. Nevertheless, as explained above it is useful for grasping the magnitude of the calculation complexity. Additionally, it is possible to employ parallelizable coding techniques and frameworks like OpenMP or simply OMP (Open Multi-Processing) C, where parallelization specific issues can be implemented and tested. These intermediate steps simulate closely the execution of the algorithm in the GPU and make the development process easier. Shown in Chart 1 below is a comparison of time required, to solve the reconstruction on an Intel Core i7-6820HQ (Quad Core CPU) using standard C (no-parallelization) vs using C-OMP (parallelized). The SAFT algorithm is inherently parallelizable (voxel values can be calculated in parallel threads) and the performance of the algorithm is accelerated by a factor which is equal to the number of threads running on different processing units. In the example of Chart 1 this is ~equal to the number of cores of the CPU (x4 times faster). Implementing reconstruction in C-OMP by applying parallelisation improved the duration from 27109 (standard C) to 5469 seconds.

The original CUDA implementation of the code on which RayFos started working was already making use of the benefits of the parallelization of the code on GPU as it can be seen in Chart 2, where we compare C-OMP code running on an Intel Core i7-6820HQ @2.7 GHz versus the code being written in CUDA on 2 different type of Nvidia GPUs the mobile grade M1000M and the workstation grade GTX1080.

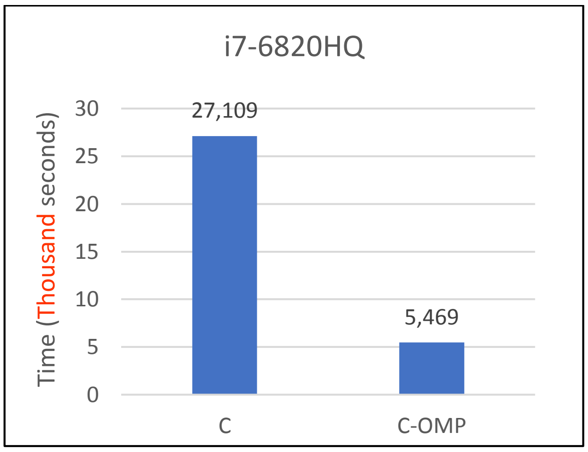

Chart 1: **reconstruction time C vs C-OMP on i7-6820HQ**

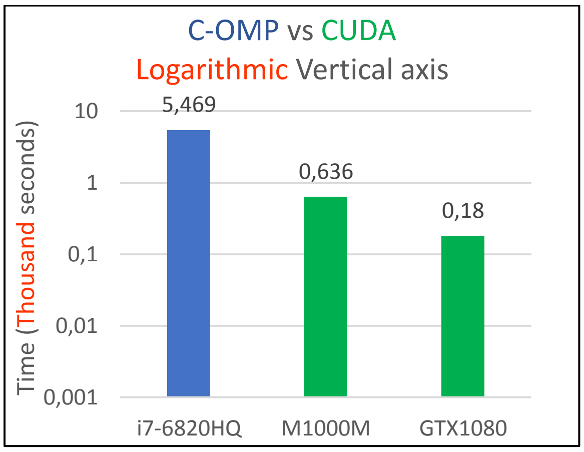

Chart 2: reconstruction time C-OMP (Blue) vs CUDA (Green) in selected GPUs

As shown in Chart 2 the intermediate C-OMP code requires 5469 seconds in a i7-6820HQ processor while the CUDA code requires 636 seconds in a mobile grade Nvidia M1000 GPU and just 180 secs in a workstation grade GTX1080 GPU; please note that Chart 2 is using a logarithmic vertical axis so as to show the dramatic increase in speed between the CPU and GPU processing times.

Once the C and C-OMP implementations/test-beds were in place we applied some basic optimizations in the code loops (see Code Optimization below). After that we were ready to port the GPU code to OpenCL which was our GPU platform of choice; this was made due to a number of advantages. Both CUDA and OpenCL are designed for solving general purpose GPU computations, but CUDA is a proprietary implementation from Nvidia while OpenCL is an open standard adopted by almost all chip vendors including Nvidia. The main thrust is that people want to be able to have portability across the various platforms. All of the major hardware vendors have agreed to the OpenCL standard, whereas CUDA targets only Nvidia hardware. Additionally, OpenCL supports not just GPUs but also CPUs, integrated GPUs (inside CPUs), FPGAs and mobile processors. Therefore, the versatility of such a Software implementation provides access to CPUs and GPUs available in laptops, desktops and many other hardware options, enabling OpenCL's ubiquity. Below a number of charts are shown depicting the increase in speed of the reconstruction using GPUs comparing CUDA to OpenCL on same devices, but also showing the interesting potential of using the integrated GPUs with impressive results on portable laptops.

Code Optimization

The improvements in speed are achieved due to a number of significant tweaks in the code; without altering the underlying methodology. Firstly, division and other unnecessary mathematical functions used within certain loops of the kernel are moved outside of it. Memory access is kept to a minimum and image buffers are introduced (instead of traditional memory buffers) so as to accelerate memory access (the effect of this change is more impressive on less powerful devices like mobile or integrated GPUs with slower memory interfaces). Finally, even the less powerful GPUs (Intel HD530: CPU-graphics) achieve very good performance (380 secs) comparable to the original CUDA code on a GTX-780 CUDA – needing 300 seconds for pure reconstruction time. The AMD R9 290 under the OpenCL platform achieved a very quick 48.59 seconds; Finally, with a Nvidia GTX 1080 GPU using OpenCL we were able to achieve a very fast pure reconstruction time of 41 seconds. Please note that figures mentioned here refer to pure reconstruction time since that part of the code has been optimised using OpenCL. The complete workflow that performs the signal processing and reconstruction is using a Matlab script that is performing a number of pre-processing tasks like signal filtering and memory copying which are not optimised yet. The impact of this tasks was trivial in the beginning when the reconstruction speed was the main performance bottleneck, but now that the reconstruction performance is significantly faster, it is important to optimise these steps also – as

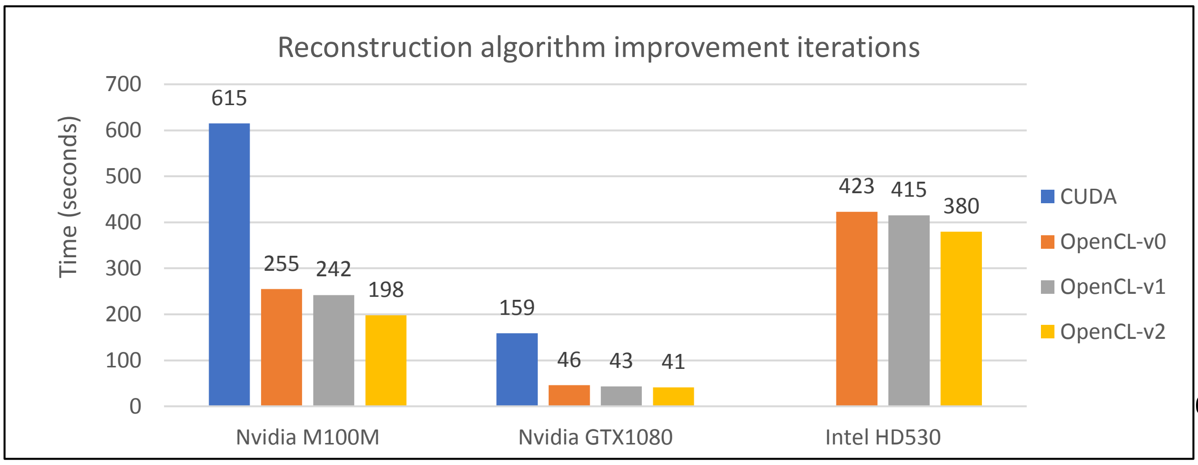

Chart 3: Durations of SAFT calculations using different vendors

discussed in Future Plans and Conclusion.

In Chart 3 the gradual improvement of the GPU code performance is shown over a number of iterations and for some representative processing devices. CUDA refers to the original code, OpenCL-v0 refers to the first implementation in OpenCL, OpenCL-v1 refers to a number of memory access optimizations and OpenCL-v2 refers to the introduction of image buffers that can perform slightly faster access than standard global memory (that is more profound on devices with slower memory).

Analysis of performance

The new version of the code in OpenCL is able to conduct the SAFT reconstruction on different vendors using the Z-slab approach - with the alterations described above. As discussed already the overall processing involves some pre-processing tasks still performed on the Matlab side. As an example, the OpenCL code using the Nvidia GTX 1080 GPU achieves a reconstruction time of 41.68 with the Matlab section taking now a considerable processing time of 20.69 seconds. These 20.69 seconds can be broken down in

- Filtering of the laser reflection lines (9.17 seconds)
- Raw signal filtering using a bandpass filter done within Matlab; (4.77 seconds).
- Another 4 seconds are consumed on data copies between memory blocks and from the host to the MEX library.

Most of these tasks can be significantly improved by GPU implementations or completely omitted (memory transfers between Matlab and MEX lib) once the whole code is implemented in a C# or C++ application which is our next goal.

GPU Charts: Comparison of reconstruction times of OpenCL on different GPUs; only pure reconstruction time.

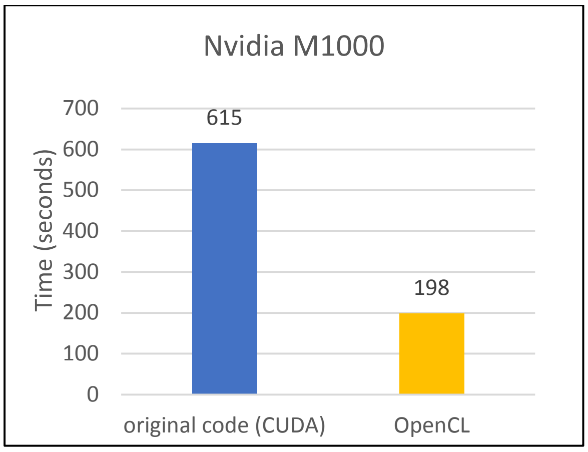

Chart 4: reconstruction time CUDA vs OpenCL on Nvidia Quadro M1000

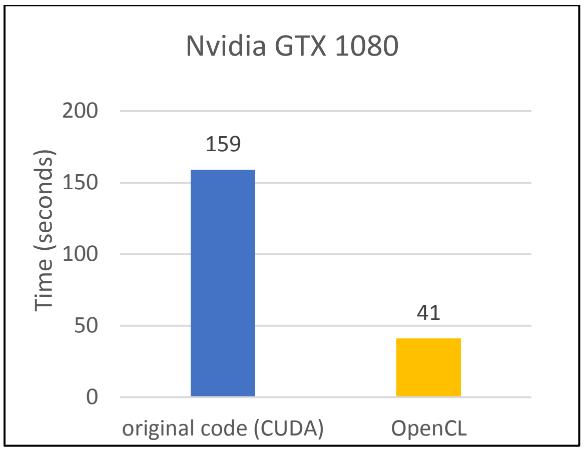

Chart 5: reconstruction time CUDA vs OpenCL on Nvidia GTX1080

As can be seen in Chart 4the Nvidia Quadro M1000M card when using CUDA was performing the reconstruction in 615 seconds; the ported code in OpenCL initially performed calculation in 255 seconds - after minor improvements this became 242 and finally with the image buffer alterations, 198 seconds. The M1000M device-typically available in Ultra-thin laptops – gets a great boost in performance by the use of image buffers (18%) because memory access in such devices is slower than the high-end workstation/gaming GPUs like the GTX1080. Similar behaviour is observed for Intel graphics GPUs, like HD530, which are embedded in (mostly mobile) Intel CPUs.

On a Nvidia GeForce GTX1080 using CUDA the reconstruction time was 159 seconds; initially in OpenCL this time was 46, after minor improvements 43 and finally after image buffer alterations 41 seconds (see Chart 5).

The speed up using OpenCL instead of CUDA described above exhibits a dramatic speed up; whether that be on the Nvidia M1000M or the GTX1080.

With the ultimate goal of reconstructing an image in real time for a patient any alternative solutions that are computationally more efficient would enhance the uptake and success of the application. With this in mind we next considered a B-frame approach. This would be a great improvement for three reasons:

1) This would enable a live preview of the region of interest -> from here we could show the image shaping up from the area already scanned.
2) If multiple wavelengths are used in scanning, it is also possible to reconstruct the individual B-frames and generate a Multispectral image by using spectral unmixing algorithms
3) When the scan is finished the entire image would also be finished, since only the last B-frame needs to be processed for completion – this is in comparison with the previous method where the application is having to wait until the entire region is scanned and calculated before producing a reconstructed image for use.

Converting the code to the B-frame approach

B-Frame Workflow

The idea of changing the orientation of the reconstruction "slicing" from vertical (Z-axis) to horizontal (along the direction of the step-wise scan axis) was attractive for the reasons explained above. We soon realized that the GPU kernel needed no changes, we only had to modify the blocks of data we were providing to the GPU accelerated function and accumulate the partial reconstruction result for the next B-frame iteration of the scan. Our implementation had in mind mainly the live-preview requirement, so we decided to call the reconstruction function for each new B-frame acquired and we realized that the achieved reconstruction time per B-frame was faster than the acquisition time of the B-frame, while keeping the same resolution as in the original Z-Slab setup. On a mobile workstation the Nvidia Quadro M1200 performed well with a speed of 140 milliseconds reconstruction time per B-frame and 40 milliseconds band-pass filtering time per B-frame. On the Nvidia GTX1080 we obtained a very quick reconstruction time per B-frame of 52 milliseconds and 14 milliseconds band-pass filtering time per B-frame (please note here that Band-pass filtering is not yet GPU optimized and depends on the Matlab implementation). The virtual sensors remain equal to the number of steps we take along the algorithm's grid positions. The resolution is kept the same; we use the axial direction with dz = 4 and along the lateral direction ds = 20.

We also had to modify the code so as to filter the signal per B-frame rather than the entire data set, without any motion correction being applied due to movement of the subject; in the B-frame approach the GPU accelerated function is called more often (for every B-frame ~500 times in our typical setup) but with a much smaller amount of data being transferred from host to device on each call. In this case the summing of the data is calculated along the y-axis; the perception is rotated 90 degrees. The kernel is altered to receive a set of data representing a B-frame rather than a z-slab – this can be thought of as a region of interest along the y-axis as shown in Figure 2 below:

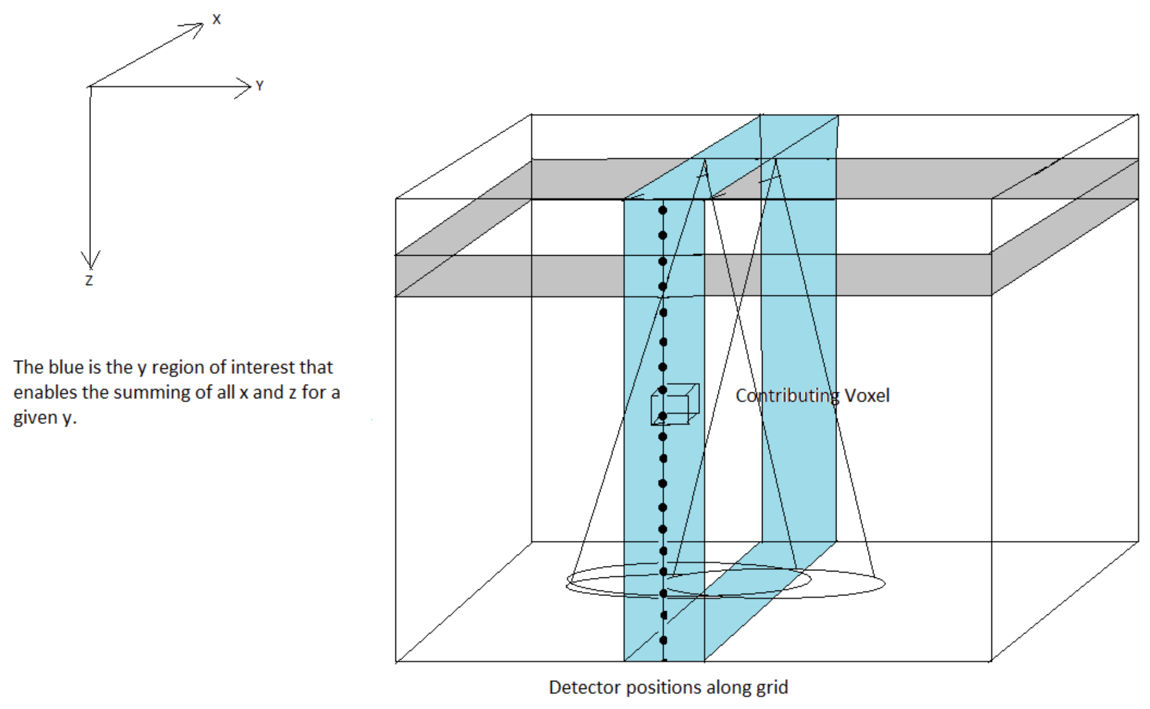
X
Y
Z
The blue is the y region of interest that enables the summing of all x and z for a given y.
Contributing Voxel
Detector positions along grid

Figure 2: B-Frame reconstruction

Acquisition protocol in the B-Frame Workflow

Since the reconstruction time per B-frame is faster than the acquisition time needed for one B-Frame, the time needed for image formation depends on the acquisition protocol itself. Improvements in the acquisition protocol, therefore, have a direct impact on the image formation time.

In the raster-scan approach a B-frame is formed by scanning the detector along a line in one direction at constant speed of stages. After one B-frame has been recorded, the data is transferred, and the stages are setup to move in the opposite direction (see **Figure 3**). We analysed the idle time between two individual B-frames with respect to its factors:

- A typical FOV measures 6mm x 6mm and includes 301 x 301 data points and recording approximately 1500 data points that with 12bit. Thus, a typical data size per B-frame is ~70KB. Data transfer rates for the used DAQ are specified with 200MB/s, which results in a data transfer time of ~3.2ms.
- The measured idle time between the last measurement of one frame and the first measurement of the following frame was measured to be ~67ms.
- Since the data transfer rate of the DAQ is much faster than the idle time, the idle time is related to the stages movement. The data transfer rate of the DAQ is not the limiting factor but the time needed for deacceleration of the stages, setup of new scan parameters for the stages and the acceleration of the stages in the opposite direction.

Hence, the idle time between two different scanning directions cannot be accelerated by improved data transfer from the DAQ (for example streaming methods with buffers) but are limited by the inertia of the stages and the scan head.

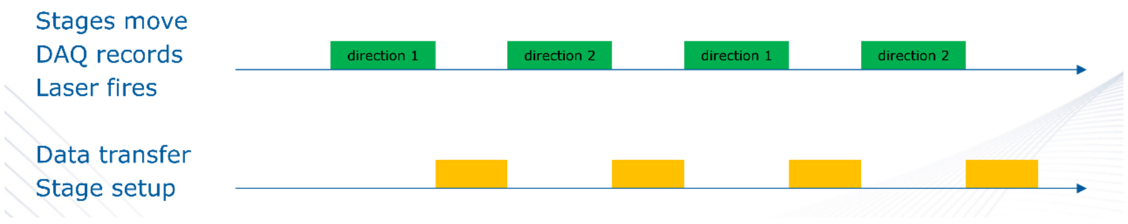

Figure 3. Acquisition protocol for the raster-scanning approach. Recording is interrupted by idle time, where data transfer happens, and the stages are setup to move in the opposite direction.

B-Frame approach reconstruction image progression

In Figure 4 and Figure 5 below the first B-Frame and the 39th B-Frame of an example dataset is shown; the live preview enables image reconstruction immediately, and when the final frame is scanned and reconstructed, the image is complete.

Regarding the live preview capability, there are alternate ways that the image may be presented to the user (mainly for reasons related to the confirmation of a correct acquisition):

- As a MIP projection of the whole volume that has been scanned so far (like in the following figures)
- Or as a MIP projection of a moving “slab” of B-Frames that follows the scan progress

We are currently in the process of applying the proof of concept demonstrated here, by integrating the B-frame reconstruction into the raster-scan acquisition. We are going to completely remove the Matlab code from the workflow so as to speed up performance of band-pass filtering and avoid unnecessary slow memory copies on the Matlab interface (C++ to MEX wrapper calls).

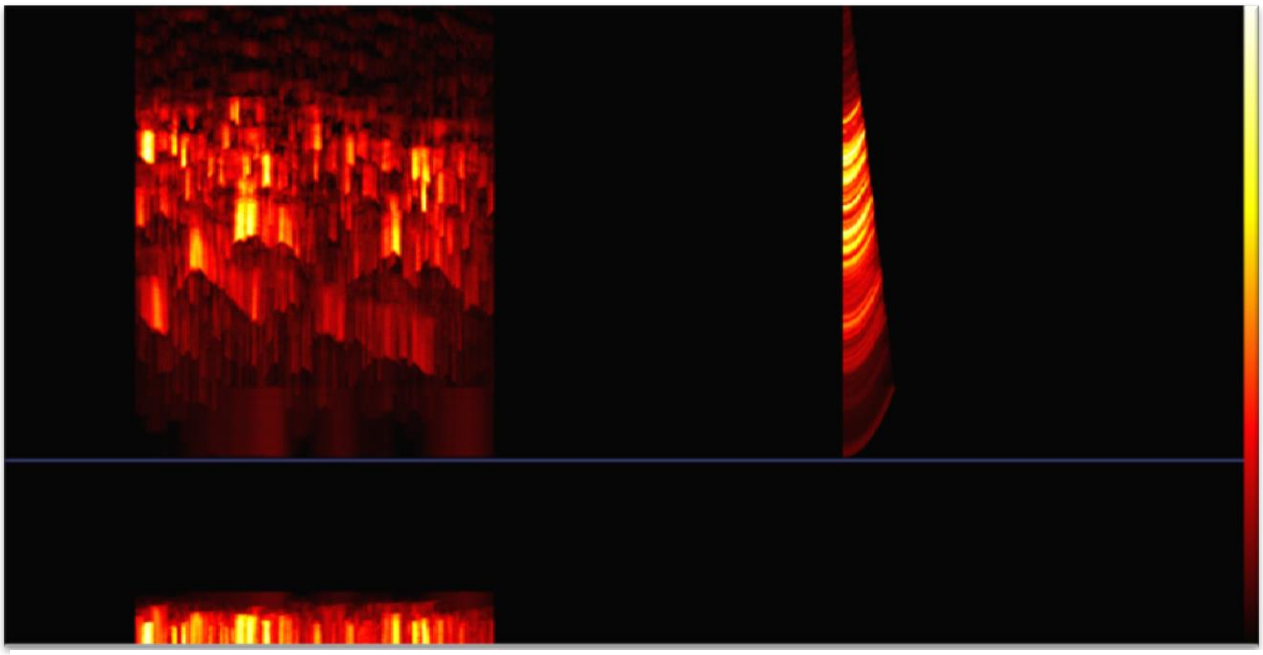

Figure 4

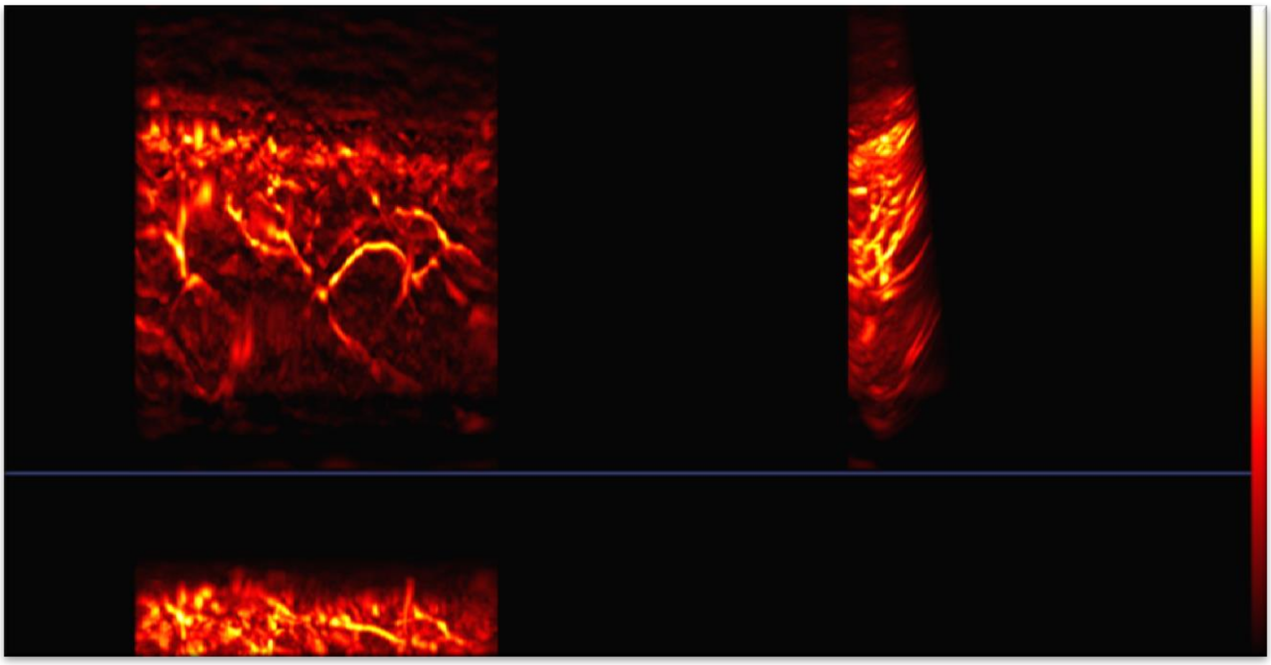

Figure 5

The images above show how the image gradually develops compared to the first B-frame – more detail and features are consolidated by the algorithm as it processes the data.

Future Plans and Conclusion

We have started with a code-base on Matlab and CUDA, that applied the SAFT algorithm to a set of data representing a patient’s ROI, which was already fast. We have then ported the CUDA code to OpenCL – with all the relevant benefits now being available. In this process we went through some intermediate steps of implementing the algorithm in C and C-OMP for analysis and debugging. The acceleration achieved which is roughly 4 times faster than the original CUDA code, is already available in several sites where the RSOM devices are installed.

Finally, we changed the approach from the Z-slab to the B-frame approach, enabling a product that can produce a live preview in practice along with the capability to provide a reconstructed volume almost immediately after the end of the raster scan. As we proceed towards a new raster-scan acquisition code that will move away from the current Matlab based workflow, we will also move the remaining small tasks (e.g. band-pass filtering) to the GPU; optimising the approach as far as possible.