# Semi-automatic calculation of blood flow through aorta

Author: Christian P. V. Christoffersen [cpvc@cs.au.dk]
Collaborators: Martin Kristensen [martinsk@cs.au.dk] and Lau Brix [lb@mta.aaa.dk]

***Abstract*—This paper presents a semi-automated method for analysing and calculating the blood flow through aorta over time. The calculations are based on MR modulus and phase images. The method utilizes the Cornelius/Kanade registration technique to compensate for the inherent motion of the heart. It uses the k-means clustering algorithm to group flow with the approximated same shaped graph. Finally it selects the group containing aorta and calculates the total blood flow through aorta over time. The method utilizes CUDA to boost performance making it possible for an operator to use it while the patient is in the MR scanner.**

***Index Terms*—semi-automatic, blood flow, aorta, magnetic resonance imaging (MRI), modulus image, phase image, registration, cluster analysis, parallel implementation (CUDA).**

## I. Introduction

THE following work has been done as part of the course: Parallel computing for medical imaging and simulation. The course was taught the fall of 2008 at the Department of Computer Science (Daimi), University of Aarhus. The work described in this paper has been done in collaboration with the MR Research Centre, Aarhus University Hospital, Skejby.

Ischemic heart disease (IHD) is a major cause of morbidity and mortality in western countries. In this context cardiac magnetic resonance (CMR) is a well established imaging technique for quantification of cardiovascular blood flow in the large vessels surrounding the heart [1]–[3]. Information about blood flow can be used to determine stroke volume, retrograde flow and for general evaluation of the heart valves.

A specific value of interest is the volume of blood that flows through aorta over time. This value is normally described by a graph, showing throughput in an R-R interval (cardiac cycle).

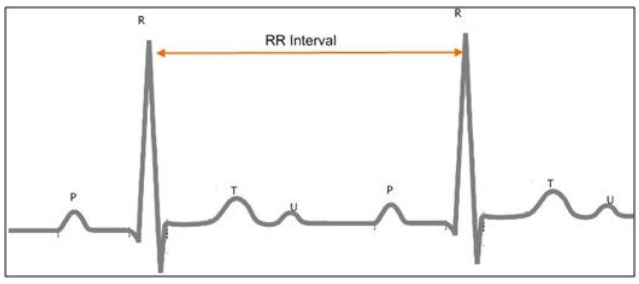


Fig. 1. Illustration of an R-R interval [11].

The common approach of measuring the flow utilizes non-isotropic 2D slice planes. An operator positions the slice during scanning. The slice must be placed perpendicular to the blood vessel that is to be analysed for the analysis to give the right results. However, a flow analysis must be carried out after the scanning has taken place. This step requires the attention of a physician who must draw the contours of the vessel under investigation in each of the 2D slices. The physician's work takes in average about 9 minutes.

If this blood flow analysis could be performed automatically the previous mentioned step could be omitted and precious time would be saved. Another benefit of doing it automatically is that the result of the analysis will always be the same whoever uses it. A problem with drawing the contours manually is that nobody does it exactly alike, not even trained experts [4]. Instead the automatic analysis will always generate the same result when applied to the same input.

If the automatic analysis could be run by the operator while the patient was in the scanner, it could be used to verify that the slice has been placed correctly (perpendicular to the blood vessel), hereby omitting the need for scheduling a rescanning of the patient.

The purpose of this article is to construct a system that does an automated flow analysis. The system needs to be as fast as possible hopefully being useful for the operator while scanning.

The method proposed uses the Cornelius/Kanade registration algorithm to make corrections for movement of the heart and the large thoracic arteries surrounding it. It uses a k-means clustering algorithm to do a flow analysis, hereby grouping the flow graphs into clusters. In the end it computes the flow in the clusters, selects the cluster containing aorta and outputs the flow graph for this cluster. The flow analysis used takes the same approach as Mouridsen et al. [4]. But instead of using it to analyse the arteries in the brain, we use it on the large thoracic arteries surrounding the heart.

This first section introduces and motivates the project. Section II describes our method. The next section (III) how we have implemented and optimized it for best performance. Then results, problems and future work are discussed. Finally the conclusion. The last part lists appendices and references.

## II. Method

The method can broadly be listed as follows. Each item on the list will be fully described in upcoming subsections.

Method:

- Acquire and store data
- Load and reformat data
- Flow analysis
  - Registration
  - Cluster analysis (grouping based on flow graphs)
  - Total flow calculations of each cluster
- Selecting the right cluster

### A. Acquiring and storing data

Images were acquired using a Philips MR scanner, which stores the data in the par/rec research file format. This format uses integers (16bit unsigned short) to store monochrome intensities. The MR scanner is set to record continuously and captures an R-R interval in approximately 5 minutes. The output data is a sequence of 2D slices through the heart at a rate of about 25 pictures per R-R interval. Each slice contains a set of images, a modulus and a phase image. A modulus image captures morphological/anatomical information by showing the hydro and lipid densities ranging from black to white. A phase image captures movement of protons, which is interpreted as flow. The phase image also shows the direction of this flow. Being white means flowing into the screen and black towards you. Gray is when the flow is zero. The exact MR scanner and technique can be found in appendix A.

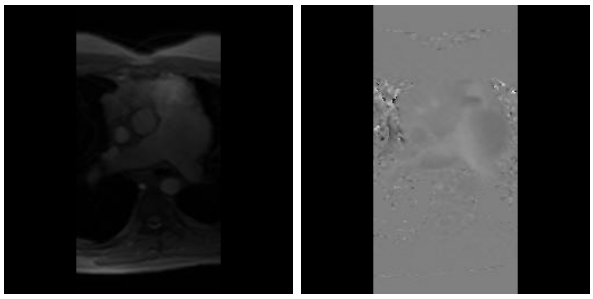

Fig. 2. Examples of: left a modulus image, and right a phase image.

### B. Loading and reformation the data

The data is then loaded and converted into an internal format that is suitable for the upcoming flow analysis (the loaded data is in 16bit unsigned short while the internal format is in single precision 32bit floating point). The intensities are normalised in the interval from 0 to 1, which makes it easy to visualize. Keeping the data in single precision floating point values makes it easier later on to execute some or all of the calculations on a GPU.

### C. Flow analysis

The flow analysis is the central part of our work. It consists of the following parts:

- Applying a **registration** algorithm to remove the inherent motion of the heart during the cardiac cycle.
- **Cluster analysis** which groups similar flow graphs.
- Finally, **calculate the total flow** in each cluster.

The registration algorithm is run on the modulus images to generate a velocity field describing the motion. This velocity field is then applied to the phase images allowing each pixel in the phase image sequence to be treated as a **flow graph**. The flow graph can be described in terms of a formula F where subscript x and y identify the pixel and t the time/slice to be indexed. The output of the formula is the intensity of the pixel.

$$F_{x,y}(t) = i \tag{1}$$

An example of a graph generated from pixel 3,4 contains value equal to intensity of pixel 3,4 in phase image 1 for t=1, intensity of pixel 3,4 in phase image 2 for t=2, and so on.

The resulting velocity field is applied to the phase image sequence to minimize the movement in the sequence. There has to be as little movement as possible for the cluster analysis to generate the right results.

*1) Registration:* A registration takes two images as input and produces a velocity field which describes the movement that has occurred from the first images to the second one. We call the first image the **reference image**, and the second the **registree**. If the input images are in two dimensions, the result is a vector field of the same size as the input, but with a 2D velocity vector per input pixel.

There are many different algorithms and techniques for doing a registration. We have studied two: Horn/Schunck [5] and Cornelius/Kanade [6]. As we had previously worked with Horn/Schunck, this was our first choice. This technique worked to some extend, but it turned out to generate some unwanted artifacts. The artifacts were generated because the Horn/Schunck registration is based on intensity preservation. After this discovery we turned to the Cornelius/Kanade algorithm which is an extension of the Horn/Schunck algorithm but does not depend on intensity preservation. We have found this method to suit our needs.

Both methods solve the problem implicit (in iterations) and have a termination criteria in terms of a threshold value. This value describes how large the difference in movement can be between the reference and registree images on a per pixel basis.

In each iteration the algorithms compare each pixel of the images in turn. When looking at one pixel the neighbouring pixel's intensities must be read and compared. This is illustrated in figure 3. The pixels on the border of the images do not have eight neighbours. This is not a problem as the registration does not include these pixels in its calculations.

| | | |
|---|---|---|
| 8 | 4 | 5 |
| 3 | 0 | 1 |
| 7 | 2 | 6 |

Fig. 3. 3x3 pixels [12]. The pixel with index 0 is under processing. All the other pixels are neighbours of pixel 0.

In order for the algorithm to find the global solution the input is first down sampled, and the registration is run on the down sampled data. The result is then up sampled and used as a starting point for solving the original problem. This can be done recursively hereby allowing the input size to be minimal when the algorithm is started. Solving the problem this way guaranties a global solution.

Down sampling takes an image and produces a new image one quarter the size (divided by two in each dimension) and converts four pixel intensities into one by interpolation.

Up sampling of the velocity field is done by making a field that is four times larger than the input and copying and scaling the vectors.

We use this in the registration: First both the reference and the registree images are down sampled to one quarter of their original sizes. This is done recursively until the size is minimal. Then a solution for the down sampled version is found, returned, up sampled and used as a approximate result for finding the solution to the problem one recursion step up the stack. This continues until the last recursion call returns containing the result.

To generate the result of the entire image sequence the registration algorithm is run twice in different ways. The following is an example of this process to illustrate how the method works. The example processes an image sequence with 5 slices, while a normal images sequence has about 25 slices capturing the cardiac cycle.

First the registration algorithm is executed for each pair of phase images sitting next to each other. This generates four velocity fields: 1-2, 2-3, 3-4 and 4-5.

The velocity fields are then added together to produce the wanted motion correction. They are added by summing the per pixel velocity vectors. Ex. if image 1 in the sequence is to be the reference image for the entire sequence, the following sums are calculated as approximate results:

1-2 = 1-2
1-3 = 1-2 + 2-3
1-4 = 1-2 + 2-3 + 3-4
1-5 = 1-2 + 2-3 + 3-4 + 4-5

Fig. 4. Example of accumulating four velocity fields with image 1 as the reference image

Finally the added fields are used as approximated result for running the registration algorithm and calculating accumulated velocity fields. This last step makes sure that the results are as correct as possible. It gives a better result and is a sort of re-registration for error correction.

In the example above the first image was used as the reference image for the whole sequence, but to get better results it is best to use the central image in the sequence. Ex. images 3 if the sequence has 5 images. This minimizes the number of velocity fields that needs to be summed, which hereby alters the original images as little as possible. Using the central image results in having to do the registration on the first part in the reverse order. It will look like this:

3-1 = 3-2 + 2-1
3-2 = 3-2
3-4 = 3-4
3-5 = 3-4 + 4-5

Fig. 5. Example of accumulating four velocity fields with image 3 as the reference image.

To examine the results of the registration we visualize them using difference images of the modulus image sequence. A difference image is generated by taking two images and subtracting their pixel intensities (running from 0 to 1) divide this by 2 and adding 0.5. The resulting image is gray (0.5) when the input images are equal, and white or black when they differ.

We generate the following three difference images to visualize the result of the registration. The first image is called the **original difference image**. This image is calculated on the basis of the original unaltered pair of images that are to be registered and shows the motion between the two, ex. between image 1 and 5. The next image is called the **registered difference image**, which is the same as the first but with the registered images as input.

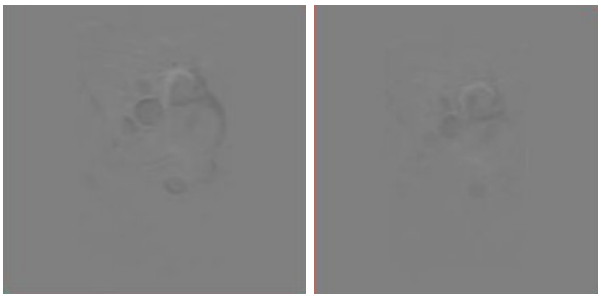

Fig. 6. To the left the original difference image and to the right the registered difference image.

Finally the third difference image is generated between the original and the registered difference images. We call this the **corrected difference images**.

Fig. 7. An example of a corrected difference image.

The corrected difference image shows how much the original images have been corrected by the registration, and gives an intuitive insight as to where the correction has been made.

*2) Cluster Analysis:* Cluster analysis or clustering as defined by wikipedia [7]:

> Clustering is the classification of objects into groups (called clusters) so that objects from the same cluster are more similar to each other than objects from different clusters.

Here we wish to group flow graphs that have the approximate same shape. We have defined the flow graph to be a per pixel graph in the time dimension of the phase image sequence, as defined in formula 1. The cluster analysis is performed on the motion corrected phase images which have a size of 256x256 pixels. This gives a total of 65.536 graphs to be processed. To do the cluster analysis we use the k-means clustering algorithm, which treats each flow graph as an x dimensional vector, where x is the number of slices in the captured image sequence. The graphs are grouped into clusters using the squared length of this vector.

$$G(x,y) = \sum_{t=0}^{n} F_{x,y}(t)^2 \tag{2}$$

The k-means clustering algorithm [8]:

- Choose the number of clusters, k.
- Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.
- Assign each point to the nearest cluster center.
- Recompute the new cluster centers.
- Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn't changed).

Instead of randomly generating the clusters or centers we started of with predefined cluster centers to have absolute control of the grouping. We did this by inserting the cluster centers directly. The algorithm will run until there is no further movement of neither the cluster centers nor the cluster groups. This is usually obtained in about 4-5 iterations.

We call the result the **cluster map**, which is an array the same size as the images, but contains a number per pixel. The number identifies which cluster the pixel belongs to.

Fig. 8. Example of the cluster map. 10 clusters are visualized, each with a different colour.

Figure 9 shows examples of binary extracts of the cluster map. Each image shows a cluster. The black colour marks pixels belonging to the cluster, and the white those that do not.

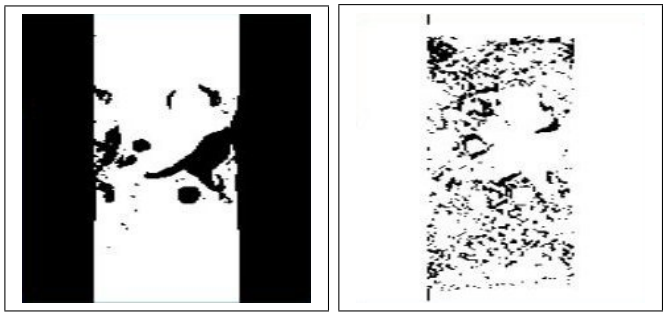

Fig. 9. Example of two different clusters from the same analysis. The one on the left groups all the graphs being nearly zero in all image slices and the right one clusters noisy graphs.

Be aware that the image in figure 8 and the images in figure 9 are generated using different parameters and therefore the clusters are not exactly alike.

*3) Calculating the total flow:* The calculation of the flow is done per cluster and is done as follows. The cluster center of each cluster defines the mean flow graph of that cluster. Therefore we can multiply the cluster center by the number of flow graphs to produce the total flow in the cluster. When this is done it must be converted back to the original unsigned short format and be multiplied according to the volumetric information in the par/rec file.

All cluster flow values are not needed, but they are all computed allowing us to know for sure how much time is at worst spent on this calculation. In worst case all flow graphs are in the same cluster.

### *D. Selecting the right cluster*

To select the right cluster, the one including aorta, we use a manual process as of now. It does not make much of a challenge to identify the cluster containing aorta, and as the best clustering results only generate about 8-10 clusters the process is not time consuming. If too many cluster centers are used, aorta is fragmented into more than one cluster.

The following is an example result of the cluster analysis showing the cluster containing aorta.

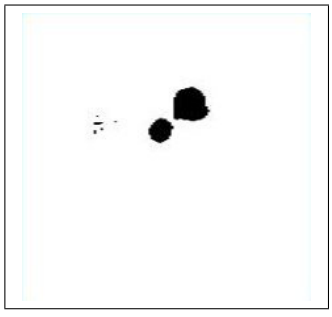

Fig. 10. Cluster containing aorta.

As can be seen in the figure, both aorta and one of the heart valves have been clustered together. This is not preferable and will be discussed further in section IV.

## III. IMPLEMENTATION

To implement the method we first implemented the entire system for execution on the CPU. Then we made time measurements and analysed these to find bottle necks in the program. The parts that proved to be bottle necks were reimplemented for execution on the GPU and optimized. As we had build the system around interfaces it was easy to plug in new implementations of the different algorithms.

To get quickly started we borrowed code for parts of the system: The par/rec loader and an implementation of the Cornelius/Kanade algorithm for execution on the CPU. The section Acknowledgment lists the people who originally wrote this code.

The following list shows the initial time measurements for the CPU version. All time measurements in this paper have been measured using an Intel Core Duo 2.66 G.Hz with 2 GB RAM and a NVIDIA 8800 GTX graphics card with 768 MB RAM.

| job | time (sec) | percent |
|---|---|---|
| loading from file | 1,312 | 4,18% |
| conversion from shorts to floats | 0,031 | 0,10% |
| registration | 27,250 | 86,73% |
| summing velocity fields | 0,406 | 1,29% |
| cluster analysis | 2,403 | 7,65% |
| calculating flow | 0,016 | 0,05% |
| total | 31,418 | 100,00% |

Fig. 11. Timetable of CPU implementation.

So the registration had to be boosted. As this is an algorithm that is easy to express for parallel execution, we wanted to execute this on the GPU. This was done using the CUDA platform, provided by NVIDIA.

*1) Basic description of CUDA:* The Compute Unified Device Architecture (CUDA) is an extension to the C programming language, which enables the programmer to use the NVIDIA GPUs for general purpose (GPGPU) programming (this only apply for NVIDIA 8000 or newer GPUs).

The extension includes special syntax. The syntax allows programmers to specify that a function is to be executed on the GPU. A function executed this way is called a kernel. In CUDA the kernels are launched with parameters that describe block and grid size. Threads are executed in blocks which again are wrapped in a grid. When a kernel is launched the grid is set for execution. When executing the grid each block is scheduled one by one and 16 threads from the block are executed simultaneously by the hardware in what is known as a half warp. The execution order of the blocks and warps is non deterministic and can be different every time the program is executed. The execution and scheduling of threads are done in hardware, which runs very fast. The extension also includes special keywords which can be used inside kernels to access thread, block and grid dimensions and identifiers.

*2) Getting optimal performance from CUDA:* To get the optimal performance from the GPU through CUDA we need to overcome the memory latency limitation imposed. The memory bus on the graphic cards has a high bandwidth but also a large latency when fetching from global memory. When programming in CUDA one needs to be aware of the fact that memory fetching must be done in correspondence to the alignment and coalescing rules as described in [9]. The alignment is done automatically by the compiler when using basic types like int or float. To get coalesced memory access each of the threads must index the memory as if the memory is to be block copied for all threads. The alignment and coalescing constraints allow the GPU to effectively hide the memory latency.

*3) Registration in CUDA:* The main thing to optimize for better performance was the registration implementation. This is also the sort of problem that GPUs originally were designed to do best, image processing. The CUDA implementation processes each pixel in the images as a separate thread. This gives coalesced memory access so the latency is hidden. As described in section II-C1 each pixel needs to read the neighbouring pixels. To do this fast we furthermore used 1D texture caching on the data. Here we could have used shared memory but the texture cache proved better results.

*4) Cluster analysis in CUDA:* To speed up the method even more we also reimplemented the cluster analysis in CUDA. This problem is a bit more complex to implement and execute in parallel. What makes it complex is that we need to recompute the cluster centers in each iteration.

It has been implemented via a divide and conqueror approach based on a tree like structure as follows. X number of pixels in the cluster map are assigned to be summed per thread. Each of the threads results are saved into another map. This map is X times smaller in size than the original, hereby making the problem X times smaller. This is done recursively on the result until a result equivalent to one pixel has been found. Although this method does not use coalesced memory access, uses a lot of memory and wastes computation cycles by adding zeros together, it is faster than the CPU version.

*5) Running time when using CUDA:* The following table shows the execution times for the project running with GPU enabled implementations of the registration and the cluster analysis.

| job | time(sec) | percent |
|---|---|---|
| loading from file | 1,328 | 13,05% |
| conversion from shorts to floats | 0,031 | 0,30% |
| registration | 7,717 | 75,81% |
| summing velocity fields | 0,434 | 4,26% |
| cluster analysis | 0,660 | 6,48% |
| calculating flow | 0,010 | 0,10% |
| total | 10,180 | 100% |

Fig. 12. Timetable of GPU implementation.

To summarize: The bottle neck has been reduced and the complete method now only takes 10 seconds to complete. This is about three times faster than the original CPU version.

## IV. Problems

### A. Artifacts in the aorta cluster

As can be seen to the left of aorta on figure 10 there is noise in the cluster. This is not a problem if the last part of the method requires manual selection to adjust the cluster. But if the method is to be fully automated this noise must be removed. We believe that these artifacts are generated by the scanning technique used but are not certain of their origin.

### B. Aorta and heart values in the same cluster

As described in section II-D both aorta and one of the heart valves have been clustered into the same cluster. As with the noise mentioned above this is not a problem when the last part of the method is done manually. We currently have no solution to this problem.

### C. Registration artifacts

Problems around the borders of the image were spotted during the implementation and test of the registration algorithm. It seems the registration draws some of the black areas that are not part of the data into the image as it computes the velocity fields. This results in the black areas flowing into the data from the sides. We hope this is solved by clipping the images before running the algorithm as described in section VI.

### D. Deforming the phase images

When deforming the flow images by the velocity field the volume in the flow images is changed. This will give a wrong result if not included in the calculations of the final flow. We suggest that the intensities of the phase images are adjusted for the correct flow to be calculated in the end.

## V. Results

### A. Correctness of the calculations

As aorta is not clustered by itself, the automated flow calculations give the wrong output. Instead we have plotted the flow graph for the cluster center of the cluster containing aorta. The graph is shown in figure 13.

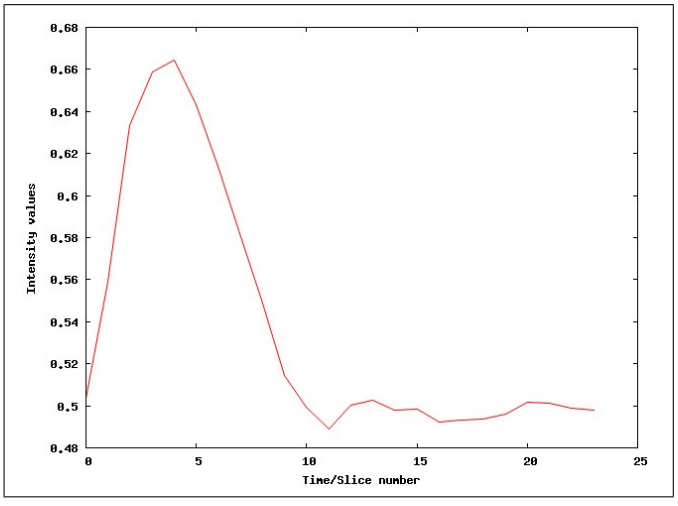


Fig. 13. Graph visualizing the cluster center of the cluster containing aorta.

While the values in this graph have not been adjusted for the volume information in the par/rec file, it can be seen that the flow graph resembles an R-R interval. As this was approximately the graph we had expected, this leads us to believe that we are on the right track.

### B. Is it fast enough?

We have had no requests from the clinic on a time bound for the process, so to answer this question we need to try it out in the clinic.

## VI. Future work

Questions and ideas have developed as we have build and implemented the method. The next subsections list some of the better ones.

### A. Segmentation

The cluster that we are interested in groups both aorta and one of the heart valves. Maybe these could be divided by running a segmentation algorithm, ex. the balloon algorithm as studied during the course.

### B. Decreasing size of the input images

*1) clipping the image:* The images are 256x256 pixels in size, but only 128x256 of these actually contain recorded data. The pixels not containing data should be removed prior to all other work as this will cut the running time in about half. Special care should be taken when the images are no longer a square.

*2) artery images:* Beside the modulus and phase images the scanning can also provide an artery image at no extra time cost. The artery image holds information about where blood vessels are located. Because these areas are the only regions of interest, this information could be used directly to clip the images prior to running the method.

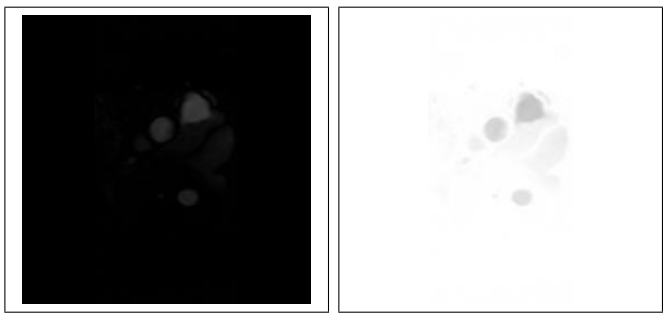

Fig. 14. Example of: left an artery image and right the same image inverted.

We suggest that one or more regions in each of the artery images are located. They could, as an example, be found by using a threshold value and represented by a rectangle describing the region. Then merging the regions one by one by expanding and combining them until a final result has been found.

Finally each of the regions of interest can be analysed separately using our method. If the right region could be selected prior to the analysis, computation time could be saved resulting in a faster approach.

### C. Optimizing the registration implemented in CUDA

We are currently not reusing the data that is copied to the memory on the graphics card. When doing two consecutive registrations on pairs of images in a sequence the last image that is used in the first registration is the same as the first in the next registration. This makes it possible to reuse the data already copied to the graphics card.

While this appears to be an obvious part to optimize it has not been a priority because it only takes a small percentage of the total time spent be the registration. This happens because the registration executes many iterations on the same data.

### D. GUI for clinical use

To be able to use our implementation in the clinic, an intuitive and efficient graphical user interface (GUI) has to be build. The code as of now is only intended for research and no effort has been put into making it easy to use.

### E. Running in real-time

We have chosen to make a prototype that runs all the calculations required of the flow analysis. But we have limited the project to load data from disk as interfacing directly with the MR scanner is non-trivial and is a project of its own. Therefore the first two steps of the method describe how data is acquired, loaded and reformated. This is a quick and easy solution, which has no effect on the results of the work.

If however the method could be run in real-time as the scanner is operating, the operator could use the results directly as guidance when placing the slice. Making it in real-time means that another scanner technique should be used. The one we have used outputs all images at the same time. To make the process in real-time the scanner should output one image at the time. This way the first step of the method (computing the velocity fields between two images that is side by side) could be done as soon as possible, hereby streaming the data from the scanner through our method.

The streaming technique cannot be done for other parts of our method. When running the cluster analysis and other steps the method requires that the entire image sequence is available. But as the registration is the most time consuming step, we believe that processing the first part of our method in a streaming like manner could increase the throughput and accelerate it.

## VII. Conclusion

Although the cluster of interest includes both aorta and one of the heart valves the results look promising. First of all because it saves time when the physician is doing the analysis. He no longer has to draw the contours on each of the slices, but can get away with only drawing the contour of aorta on the cluster map.

The focus has been on making an automated method. While this has not been achieved we have a semi-automatic solution to the problem. Furthermore it is only the last part, the selection of the right cluster, that is manual.

Even though the method is not fully automated, it can be used in the clinic. The physician just needs to draw the contour of aorta on the cluster map as the final step to help the clustering.

## Appendix A
## Acquiring data using a MR scanner

The images were acquired using a whole body 1.5 Tesla Philips Achieva (R2.1.3) MRI system. The 2D MRI technique used is a conventional segmented k-space (factor = 3) navigator gated flow sequence with a non-isotropic spatial resolution of 1.41x1.41mm and a slice thickness of 5mm. The flip angle was 25 degrees. The matrix size was 256x256, and two signal averages were acquired. The 2D slices were placed approximately 3cm above the aortic and the pulmonary valves respectively. Both scan techniques used prospective triggering and the navigator was played out for 35ms at the beginning of each cardiac cycle. The 2D images were acquired in approximately 3 minutes with a navigator efficiency of 50 percent.

## Appendix B
## Code and documentation

The code written for the project can be found on the project website at: [http://www.daimi.au.dk/~cpvc/autoflow/], where this paper also can be found in different digital formats.

The code has been written in Microsoft Visual Studio (VS) 2005 SP1©with the NVIDIA CUDA 2.0 toolkit and SDK installed. The project has only been used and tested as 32bit binaries on Microsoft Windows XP© and Microsoft Windows Vista©. The main VS project is called *OpenGLTextureViewer*.

## Appendix C
## Kernel execution time charts

Figure 15 shows a graph and figure 16 a table of the execution time measured of the kernels done with the CUDA profiler. The *solveSchemeKernel* kernel executed the registration. The *combineCenters* runs the central part of the cluster analysis that sums the centers as described in section III-4. The other kernels are helper kernels that do trivial jobs.

## Acknowledgment

The authors would like to thank Thomas Sangild Sørensen and Kim Mouridsen for lectures, talks and guidance concerning the project. Karsten Noe for a CPU implementation of the Cornelius/Kanada registration and a CPU and GPU implementation of the Horn/Schunck registration. We would also like to thank Allan Rasmusson for providing code for loading par/rec files and the MR Rearch Centre on Skejby university hospital for providing us with MR images to test the system. Last but not least: Thanks to the OpenEngine.dk project for providing easy to use cross platform utilities. We have used their Timer and Convert implementation to make sure that the code can be ported to different platforms with minimal effort.

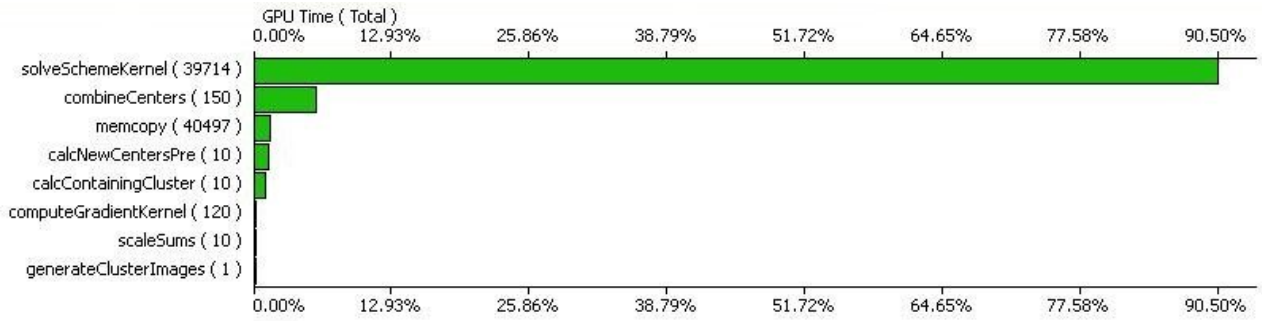


Fig. 15. Graph visualizing the time table from the CUDA profiler.

| Method | #Calls | GPU usec | CPU usec | %GPU time |
|---|---|---|---|---|
| solveSchemeKernel | 39714 | 7147770 | 520559 | 90.5 |
| combineCenters | 150 | 450966 | 491.278 | 5.71 |
| calcNewCentersPre | 10 | 95625.2 | 32.256 | 1.21 |
| calcContainingCluster | 10 | 75192.7 | 37.3 | 0.95 |
| computeGradientKernel | 120 | 10760.9 | 659.437 | 0.13 |
| scaleSums | 10 | 156.384 | 45.94 | 0 |
| generateClusterImages | 1 | 54.272 | 4.528 | 0 |
| memcopy | 40497 | 117137 | 1.48 | 1.59 |

Fig. 16. Timetable of GPU implementation using the CUDA profiler.

## REFERENCES

[1] J. Lotz, C. Meier, A. Leppert, M. Galanski, *Cardiovascular flow measurement with phase-contrast MR imaging: basic facts and implementation*, Radiographics 2002,2 2:651-671.
[2] S. R. Underwood, D. N. Firmin, R. H. Klipstein, R. S. Rees, D. B. Longmore, *Magnetic resonance velocity mapping: clinical application of a new technique*, Br Heart J 1987, 57:404-412.
[3] N. J. Pelc, R. J. Herfkens, A. Shimakawa, D. R. Enzmann, *Phase contrast cine magnetic resonance imaging*, MagnResonQ1991,7:229-254.
[4] K. Mouridsen, S. Christensen, L. Gyldensted, L. Østergaard, *Automatic Selection of Arterial Input Function Using Cluster Analysis*, Magnetic Resonance in Medicine 55:524-531, 2006.
[5] B. B. K. Horn, B. G. Schunck, *Determining Optical Flow*, Artificial Intelligence 17:185-203, 1981.
[6] N. Cornelius and T. Kanade, *Adapting optical-flow to measure object motion in reflectance and X-ray image sequences*, Proc. of the ACM SIGGRAPH/SIGART interdisciplinary workshop on Motion: representation and perception, 1986 p.145-153.
[7] Wikipedia, Cluster analysis, [http://en.wikipedia.org/wiki/Data_clustering]
[8] Wikipedia, K-means algorithm, [http://en.wikipedia.org/wiki/K-means_algorithm]
[9] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide*, 2nd version. 2008.
[10] Wikimedia commons, picture of aorta, [http://commons.wikimedia.org/wiki/File:Aorta.jpg]
[11] National instruments, illustration of an R-R interval, [http://zone.ni.com/devzone/cda/epd/p/id/5832]
[12] Todd Lawrence Veldhuizen, *Grid Filters for Local Nonlinear Image Restoration* [http://ubiety.uwaterloo.ca/~tveldhui/papers/MAScThesis/node36.html

**Christian P. V. Christoffersen** is a computer science graduate student specialised in computer graphics and simulation. He has a bachelor degree in computer science from the University of Aarhus 2007 and a degree as Bachelor of Engineering (in Information Technology) from Vitus Bering University College in Horsens 2005.