A GPU-based Fiber Tracking Framework using Geometry Shaders

Alexander Köhn [1], Jan Klein, Florian Weiler, Heinz-Otto Peitgen
Fraunhofer MEVIS, Institute for Medical Image Computing,
Universitätsallee 29, D-28359 Bremen, Germany

ABSTRACT

The clinical application of fiber tracking becomes more widespread. Thus it is of high importance to be able to produce high quality results in a very short time. Additionally, research in this field would benefit from fast implementation and evaluation of new algorithms. In this paper we present a GPU-based fiber tracking framework using latest features of commodity graphics hardware such as geometry shaders. The implemented streamline algorithm performs fiber reconstruction of a whole brain using 30,000 seed points in less than 120 ms on a high-end GeForce 280 GTX graphics board. Seed points are sent to the GPU which emits up to a user-defined number of fiber points per seed vertex. These are recorded to a vertex buffer that can be rendered or downloaded to main memory for further processing. If the output limit of the geometry shader is reached before the stopping criteria are fulfilled, the last vertices generated are then used in a subsequent pass where the geometry shader continues the tracking.

Since all the data resides on graphics memory the intermediate steps can be visualized in real-time. The fast reconstruction not only allows for an interactive change of tracking parameters but, since the tracking code is implemented using GPU shaders, even for a runtime change of the algorithm. Thus, rapid development and evaluation of different algorithms and parameter sets becomes possible, which is of high value for e.g. research on uncertainty in fiber tracking.

Keywords: Visualization, Fiber Tracking, Diffusion Tensor Imaging, GPU

1. INTRODUCTION

Diffusion tensor imaging (DTI) is an emerging technology in the neuroimaging, neurosurgical and neurological community. It is a method to identify major white matter tracts afflicted by pathology or tracts at risk for a given surgical approach. A geometrical reconstruction of fiber tracts has become available by fiber tracking based on DTI data. Since this technique has entered clinical routine, high quality and very fast reconstruction of fibers is of great value. Furthermore, by accelerating the tracking and providing a framework for fast implementation of different algorithms the effects of tracking parameters and algorithmic changes as well as advanced approaches to white matter reconstruction such as probabilistic fiber tracking can be evaluated much more efficiently.

Previous work in the field of fiber tracking on the GPU has mostly focused on enhancement of the visualization [1,2,3,4]. Also, some methods exist for reconstruction [5,6,7]. They all use the fragment processor and render the computed fiber point properties, like its position vector, to textures. This leads to an overhead when rendering these points, which can be avoided by using the geometry processor. Furthermore, this method can only compute one fiber point of a fiber per iteration, increasing required communication between CPU and GPU. In [5] the authors use the GPU solely for tracking and download the computed fiber points every iteration, sort out the finished fibers and upload points of unfinished ones for the next iteration. They also address the difference in floating-point computations between CPUs and GPUs, which is due the GPUs being not strictly compliant to the IEEE 754 standard [8,9], and provide a method to estimate the difference of the results. Therefore, we do not bother with this issue here. In [6] a stochastic tracking algorithm is implemented using fragment processors. They track a bundle of fibers from one seed point using a probabilistic approach and accumulate the fibers into a 3D texture. Using a particle engine for visualizing the fibers in a

[1] Further author information: (Send correspondence to)
Alexander Köhn
E-Mail: alexander.koehn@mevis.fraunhofer.de

dynamic way has been proposed in [7]. This technique iteratively advects particles using tensor information, which requires computation of new positions in each frame. This approach also allows rendering streamlines or tubes using the positions of the particles.

We present a novel GPU-based framework that manages the tracking and visualization of fibers. It enables interactive whole brain fiber tracking and allows for efficient implementation and testing of new tracking algorithms and parameter sets. To our knowledge this is the first approach which uses the geometry processor introduced with DirectX 10 generation graphics boards (NVIDIA GeForce 8000 series and ATI Radeon HD series). In contrast to fragment processors they provide the ability to generate new geometry and write it to dedicated GPU memory, which in turn can be visualized efficiently. Known methods to visualize the generated fibers like simple lit streamlines, more complex streamtubes or dynamic particles can all be implemented using the geometry shader without the need to write C++ code. Thus, rapid prototyping of tracking as well as of visualization algorithms is possible. The framework allows to define additional input parameters for the shaders using the network-based rapid prototyping platform MeVisLab [10].

2. METHODS

Fiber tracking is an inherently parallel task since the same algorithm is executed for each seed point, independent from each other. Using GPUs to do this is very obvious but for a long time they lacked the capability to directly generate new geometry. With a new feature of the shader model 4.0 this can now be accomplished [11]. Fundamental for this work is the geometry shader extension which introduces the geometry shader to the GL shading language. The geometry stage processes primitives, i.e. points, lines or triangles, consisting of vertices coming from the vertex stage. It allows to create new geometry and alter the incoming one, providing access to adjacent primitives if needed. Current hardware is limited to output at most 1024 floating point values, i.e. a geometry shader cannot output more than 256 4D vertices. The transform feedback extension, currently only supported by NVIDIA GPUs but included in OpenGL 3.0, allows to record the vertices, emitted in the geometry shader stage, to vertex buffer objects (VBOs). These can be rendered directly, which is a significant advantage over methods that use the fragment processor. Using fragment processors, two possibilities exist to render the points indirectly which however both produce undesired overhead. First, the computed fiber point positions need to be written to a temporary texture. Afterwards, one can render auxiliary vertices and use a vertex shader to lookup their final output position from the temporary texture in each frame. Alternatively one can copy the content of the temporary texture to a VBO and render that.

Our proposed framework can be divided into two stages, namely the geometry generation or tracking stage, and the visualization. Figure 1 gives a schematic overview of the framework. The DTI data, additional parameters and seed points are passed on to the geometry processor. It writes the resulting data, i.e. the fiber point position, direction, status flag, length, anisotropy and custom data, to a VBO. If more than one iteration is necessary to finish tracking, the VBO content is downloaded to CPU memory and the end vertices of unfinished fibers are used for the next iteration. If desired, the VBOs can be rendered after each iteration to visualize the tracking process using the active visualization method.

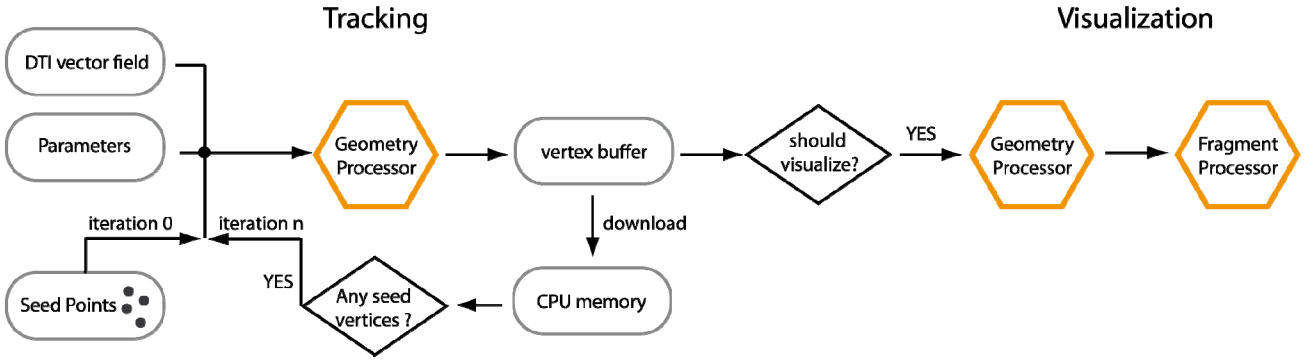


Figure 1: Tracking and Visualization stage of our proposed GPU fiber tracking framework.

2.1 Tracking Stage

The first part of the framework manages the tracking. First it has to allocate resources and setup OpenGL. Then the tracking process is initiated. If the maximal length of the fibers is expected to be higher than the vertex output limit of the geometry shader an iterative process can be activated. If the visualization of the iterative tracking process is enabled, the currently tracked fibers are rendered after each iteration using the chosen visualization method.

2.1.1 Setup

In general, the setup that enables fiber tracking on the GPU using geometry shaders involves binding a VBO, activating transform feedback, disabling the rasterization stage and enabling the vertex and geometry shaders. The VBOs are configured to record all output parameters in an interleaved manner. This avoids a dynamic allocation of VBOs dependent on the number of output parameters and thus eases the setup. For the transform feedback mechanism it is necessary to tell OpenGL which output parameters of the geometry stage should be recorded to the attached VBO. To accomplish this we use a naming convention which allows us to dynamically add parameters. All parameters that should be recorded have to start with *'out'*. The only parameters which are mandatory are the position - *outPosition* - and the direction - *outDirection* - of the fiber point. These are written to the fiber data structure in CPU memory after download. Additional parameters are not stored in CPU memory but instead sent back to the next tracking iteration. There the position is always accessible via the GLSL variable *glPosition* in the shader. The direction and additional parameters are sent to the GPU using texture coordinate units *glTexCoord[0...n]* (n is the number of available texture units which is hardware dependent). I.e. our direction output parameter is available via the *glTexCoord0* GLSL variable inside the tracking shader.

For the iterative process we need to allocate two VBOs. While one is bound as the transform feedback target we read from the second one. We allocate the two VBOs when the seed point list is updated or connected to the tracking module. If the visualization stage is enabled we have to allocate enough memory for the VBOs to hold the complete fiber bundle. Since we do not know the length of the fibers in advance it is difficult to estimate the required memory. Thus we allocate per default enough memory for 10,000 fibers with an average length of 200 points and 64 bytes per point leading to 122 MB per VBO. Of course, the buffer size is manually adjustable.

2.1.2 Non-Iterative Tracking

The seed points from which to start tracking may e.g. be generated from a user defined region of interest or from the set of voxels of the DTI image where the fractional anisotropy (FA), defined as the normalized variance of the eigenvalues, is above a certain threshold. The latter approach is commonly used for whole brain tracking. The tracking is initated by sending two 4D-vertices per seed point to the geometry shader. The fourth component, i.e. the *w* component of the *xyzw* vector, encodes the direction in which to track. In the geometry shader the sampled direction vector is multiplied by the *w* component of the vertex, i.e. a 1 leads to an unchanged and -1 to a reversed direction. The point is tracked through the DTI volume in a loop until a stopping criterion is met or the user defined number of tracking steps is reached. The upper bound for the number of tracking steps is determined by vertex output limit of the geometry shaders. Current high-end GPUs are limited to emit 1024 floating point values at most. Using a tracking algorithm that e.g. outputs 8 values per vertex such as the position and direction one can emit at most 128 vertices per input vertex. For the visualization stage we need adjacency information therefore we have to generate two additional vertices, one at the beginning and one at the end. Both have the position of the start and end point respectively. Thus one can create a fiber segment consisting of 126 different fiber points for our exemplary case. Since a seed point is tracked in both possible directions separately we can compute fibers that have at most 252 points with one iteration.

2.1.3 Iterative Tracking

If the number of generated points per fiber exceeds the number of vertices that can be emitted by the geometry shader within one iteration, or if the tracking process shall be visualized, an iterative scheme becomes necessary. The first iteration is exactly like described in the previous section. For subsequent iterations those fiber points, where the tracking in the previous iteration stopped, have to be identified and sent back to the geometry processor. This process is continued until the stopping criteria are met for all fibers.

To be able to identify the fiber points from which to proceed the tracking or when to stop tracking a fiber, we use the *w* component as a status flag. We distinguish four states:

- A fiber point marks the end of a segment and tracking should continue, i.e. the number of tracking steps was reached.
- A fiber point marks the end of the fiber, i.e. the stopping criteria were met.
- A fiber point is just part of a fiber segment.
- A fiber point is duplicated at the start or end of the fiber segment because it is needed for visualization.

Therefore we not only record the vertex' xyz position but also this status flag. The length of the tracked path, which is used as a stopping criterion, is also stored by multiplying it by 10 and adding it to the status flag. This compression saves bandwidth and graphics memory.

For each iteration, all properties such as position and direction of those fiber points, required to continue tracking, have to be resent to the geometry processor. To do so we investigated two methods:

A. Render the vertices written to the VBO, which was the transform feedback target in the previous iteration, and determine in the shader if tracking should continue per vertex by inspecting the status flag.

B. Download the data and only send the points to the GPU where tracking should continue.

For method A we have to render all vertices of VBO A, generated in the last step, and bind a second VBO B for transform feedback. The shader has to check the vertex' status flag to determine if tracking should continue. The number of vertices generated by the geometry shader can be queried from OpenGL and used to determine when the tracking should stop, which is the case if it equals 0. Alternatively one could just specify the maximum number of iterations. If the fiber bundle is needed in CPU memory to do further processing one can choose to download the VBO content. By inspecting the status flag the points can be sorted into an appropiate data structure.

In method B we download all fiber points of the last iteration to CPU memory. To be able to add the new fiber points to the correct fiber in CPU memory we keep track of the finished fibers. The status flag of a fiber point indicates if the current fiber is finished or if we have to send the last point of this fiber segment to the GPU in the next iteration. So in contrast to method A, we only send vertices to the GPU for which the tracking should proceed. Since the CPU is involved after each iteration and data gets downloaded, there is potential for parallelizing the process. The idea is that the CPU can download the data of one VBO and fill the fiber data structure, while the GPU performs tracking on another subset of fibers, writing to the second VBO. Good candidates for the two subsets are the two directions into which each seed point is tracked. The GPU and CPU get synched when the VBO, whose data has been downloaded, is bound. As shown in the result section the download and sorting into CPU memory is the bottleneckfor our exemplary fiber tracking implementation. Thus, first we will optimize our fiberset data structure and depending on that more complex tracking algorithms may be possible without performance penalties. Figure 2 illustrates this interleaved tracking. This approach proves to be about two times faster than method A. This is due to the fact that much less vertices are sent through the vertex and geometry stage, which get discarded anyway in method A. It also has the advantage that the data is already in main memory after tracking for further processing like clustering. This also allows to track a huge amount of fibers that might be too large to fit into GPU memory since the data gets downloaded every iteration and thus can be overriden in the next iteration. Consequently, this is the method of choice.

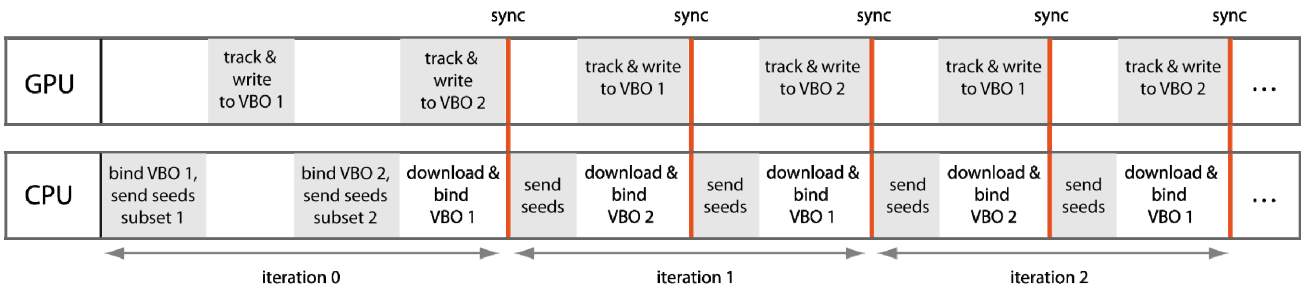


Figure 2: Illustration of method B, the parallelized tracking. While the GPU tracks and writes the results to VBO 2 the CPU downloads the contents of VBO 1, sorts it into the data structure and then binds VBO 1. CPU and GPU get synched now. The CPU sends the vertices to proceed the tracking and starts downloading VBO 2 etc.

2.2 Visualization Stage

The visualization stage can be enabled for the iterative tracking to show the progress. After each iteration the vertex buffer objects are drawn so one can see the fiber bundle growing. For this each buffer is rendered as a line strip. The geometry shader first checks if the first vertex of the line segment is an end vertex by inspecting its status flag. If not, it clips the line against the view frustum. If the line is visible it is used as a basis for the visualization method. Like for the tracking shader one has the possibility to implement custom visualization shaders. One also has access to the recorded data by using the *glTexCoord[0..n]* parameters. The input (points, lines, lines with adjacency) and output geometry type (points, line strip, triangle strip) of the geometry shader can be configured to fit the needs. Furthermore, additional parameters can be plugged into the scene graph and used inside the shader.

3. IMPLEMENTATION

Our framework is embedded into the rapid prototyping platform MeVisLab [10] which supports visual programming of Open Inventor scene graphs [12] and image processing networks. A functional unit, called a module, is instantiated and connected to other modules by the user. The implemented GPU fiber tracking module provides editable standard shaders for tracking and visualization. It also exposes stopping criteria parameters like minimal anisotropy or fiber length and common tracking parameters like the step size.

The standard tracking and visualization shaders can be changed so other algorithms can be implemented. If additional parameters are needed, appropriate shader parameter modules can be plugged into the scene graph. They are automatically available in the shaders and adjustable from within MeVisLab.

The vector or tensor dataset and a list of seed points, generated from regions of interest (ROIs) or binary images, can be connected to the module. The module downloads the dataset to one (for vector data) or two (for diagonal symmetric tensors) 3D RGB textures with 16 or 32 bit floating point precision and initializes the tracking stage. The module has one output providing the resulting fiber bundle in CPU memory for further processing and one output for visualization.

3.1 Fiber Tracking Algorithm

For comparison of GPU and CPU performance we implemented the following tracking algorithm. Of course, any other algorithm could be utilized. The algorithm uses the main eigenvectors of the reconstructed tensors. Therefore, one 3D RGB texture is required, which stores each 3D vector in the three color channels with single floating point precision. The used tracking algorithm is an extension of the basic advection algorithm [13], where the direction vector is linearly interpolated, weighted by the step size and added to the current position. We use a more advanced method that additionally weights the influence of the 8 surrounding eigenvectors by its scalar product with the current incoming direction before interpolating linearly.

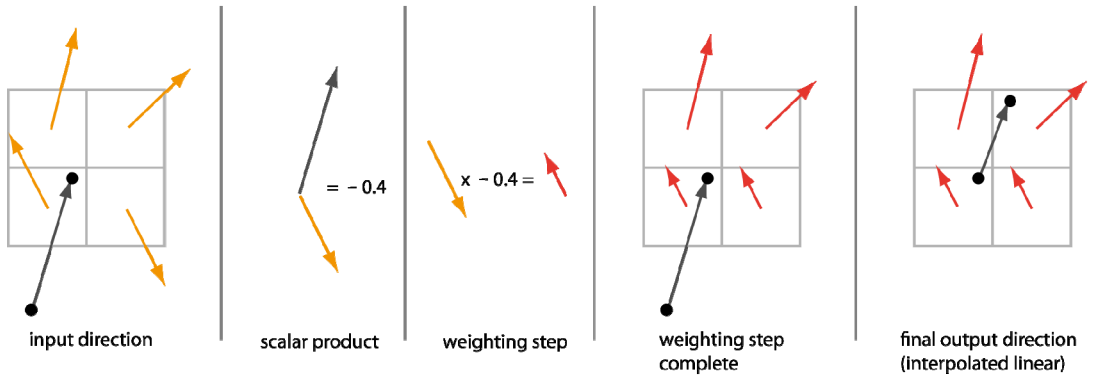


Figure 3: Illustrates the streamlined tracking algorithm in two dimensions. First, all four neighbour directions are sampled. Each is weighted by its scalar product with the incoming direction vector from the previous step. Then they are linearly interpolated and multiplied by the step size, which is 1 in this example. The resulting direction vector is added to the current position to yield the new position.

To reduce texture access the texture samples are cached in GPU registers for later reuse. This is often beneficial for small step sizes where the voxel position of the currently tracked point does not change within successive steps. For this algorithm we record 8 values per fiber point: The position (3 components), the direction vector (3 components), the anisotropy value and the status flag plus length. Thus we can at most emit 128 vertices per input vertex.

3.2 Visualization methods

We implemented three visualization methods (see Figure 4 for examples) to show the flexibility of the framework using geometry shaders.

3.2.1 Lit Streamline

To light the lines we use the method as described in [3]. To be able to compute the same normal for each fiber point, we need adjacency information thus we have to set the input type of the geometry shader to `GL_LINES_ADJACENCY_EXT`. This makes sure that the geometry shader has access to 4 vertices per line segment whereas the first and last vertex are the vertices before and after the current processed line segment.

3.2.2 Streamtube

The stream tube generation, as described in [2], is implemented by using the cross product of the view vector and the points tangent, computed from the previous and next point, to construct tube normals and triangles. To be able to construct the same tangent for the same vertex of two subsequent line segments we have to render the vertices as lines with adjacency information like it is done for the streamlines.

3.2.3 Particles

Additionally to these stream rendering methods, a dynamic particle rendering like in [7] is also possible by generating one vertex inside the geometry shader per input line segment. The position along the segment can then be animated by a shader parameter providing the time.

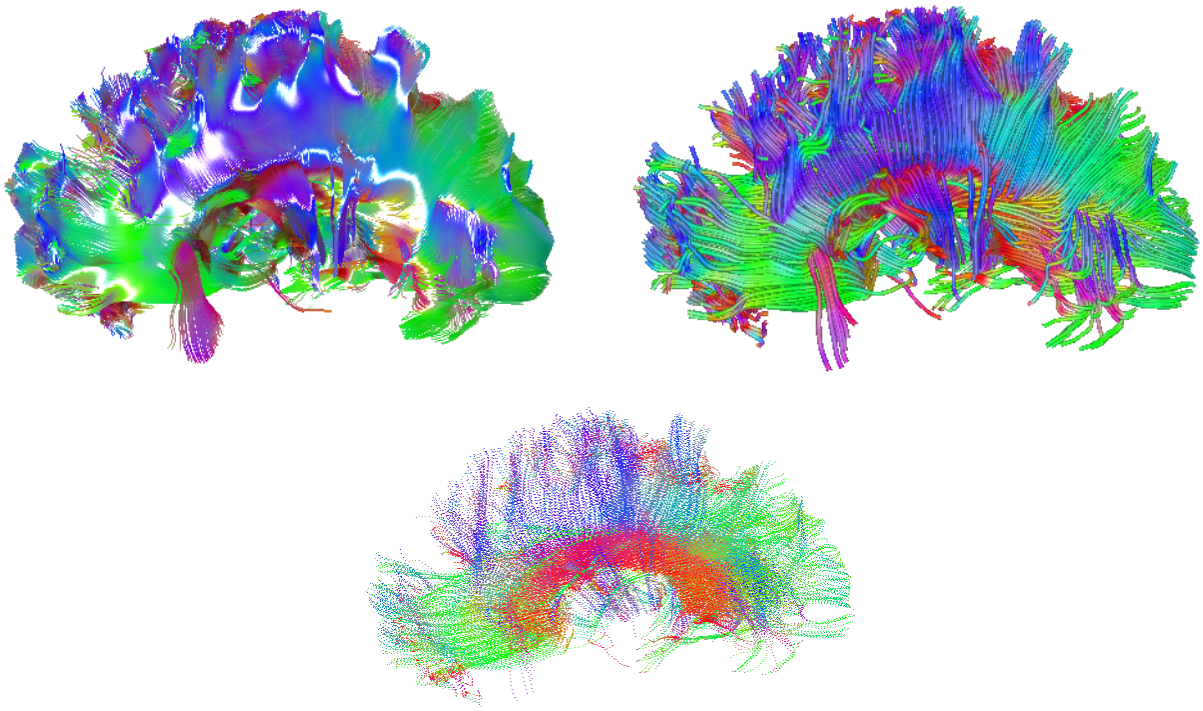


Figure 4: Screenshots of the three implemented visualization methods. Top left: very dense lit streamlines (~15,000 fibers). Top right: streamtubes (~1,000 fibers). Bottom: Particles (~1,000 fibers, each consisting of 110 points)

4. RESULTS

We have implemented the control of shader execution and setup of the GPU using C++ and OpenGL. The shaders are written in GLSL. Fiber tracts were computed using the tracking algorithm described in section 3.1 from DTI data with a resolution of 153 x 153 x 100 voxels acquired at a 1.5T Siemens MR scanner. Figure 7 and 8 give a visual comparison between GPU and CPU generated fibers. The double precision used on the CPU in contrast to 32 bit precision on the GPU and the discussed non IEEE 754 compliance [9] lead to the very small differences. We use MeVisLab to edit the shaders and visualize the fiber tracts. All tests have been performed using two systems. Details are listed in Table 1.

OS	CPU	RAM	GPU	VRAM	GPU DRIVER
Windows Vista 64 bit	**CPU 1**: Intel Core 2 Quad Q9450 @ 2.66GHz	8 GB	**GPU 1**: GeForce GTX 280	1 GB	ForceWare 181.20
Windows XP	**CPU 2**: Athlon 64 X2 Dual @ 2 GHz	2 GB	**GPU 2**: GeForce 8800 GT	1 GB	ForceWare 181.20

Table 1: Listing of hardware used in experiments.

Figure 5 shows the raw tracking performance and speed-up factors for a given number of seed points and step sizes. No setup, data uploads and downloads are included in the timings. As expected the computation time is linearly dependent on the number of seed points. Reducing the step size has a significant influence on the performance, since the caching of texture samples in this case avoids unnecessary texture accesses. If the caching is disabled, the step size has nearly no effect on tracking time, the small speed-up comes from the internal GPU cache. This can be seen in Figure 6.

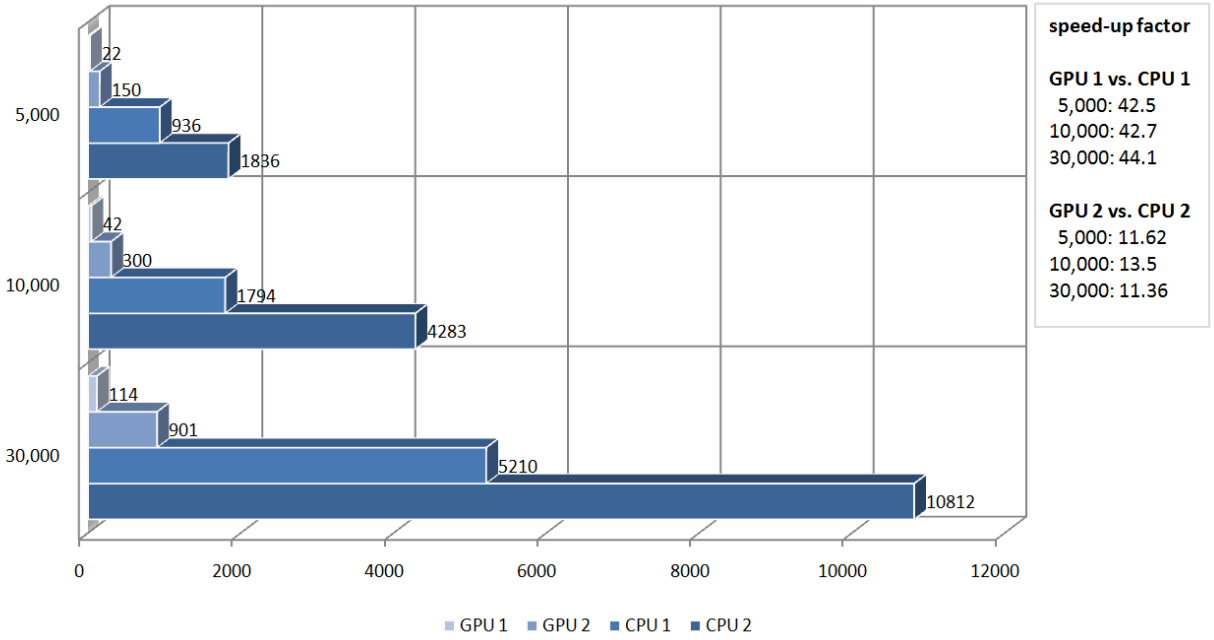


Figure 5: Comparison of CPU and GPU tracking performance for a fixed number of seed points. The step size was 1. Texture components had 32 bit floating point precision. All fibers had a length of 110 points. GPU algorithm used one iteration.

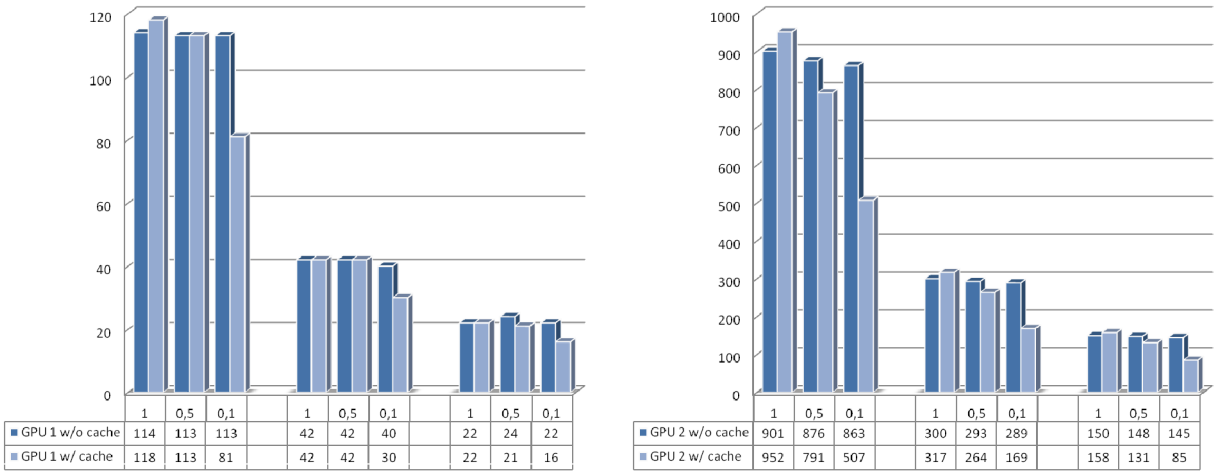


Fig. 6: Effect of caching texture samples for different step sizes for both GPUs.
Left: GeForce280 GTX. Right: GeForce 8800 GT

In our test scenario, the GeForce GTX 280 (GPU 1) is up to 44 times faster than an Intel Core 2 Quad Q9450 (CPU 1). However, since the CPU implementation of our algorithm is singlethreaded one can estimate a more realistic speed-up factor in the order of 11 to 15 when utilizing all four cores of the Quad Core processor. With our current implementation, the most time intensive process is to populate our fiberset data structure with the downloaded fiber points, as can be seen in Table 2. This is due to the fact, that this data structure has not been designed for this scenario in the first place. When activated, the speed-up reduces to a factor of about 4 compared to CPU 1, which would nearly cancel out, if all CPU cores were used efficiently. As one can see in Table 2, the pure download of the data to contiguous CPU memory only requires about 15% of the tracking time. Considering an optimized data structure for storing fibersets in the main memory, a speed-up factor of at least 10 appears realistic comparing a high-end GPU to a high-end CPU, that utilizes all cores.

#Seed Points / FA threshold	Time in ms for one iteration (GeForce GTX 280)		
	tracking	tracking, download	tracking, download, sort
30,000 / 0.4525	114	130	1200
10,000 / 0.563	42	49	500
5,000 / 0.622	22	25	210

Table 2: Time needed for tracking, downloading and sorting the data into our data structure.

For the first time we make use of geometry shaders to reconstruct white matter tracts. A fiber reconstruction using the described exemplary algorithm for the whole brain using 30,000 seed points and a maximal fiber length of 110 points is computed in one second using a mid-range consumer graphic card and in about 120 ms using a high-end GPU. This is up to 44-times faster than a single threaded CPU implementation on a high-end Intel Core 2 Quad CPU. Since the generated geometry resides on graphics memory it can be efficiently visualized even while the tracking takes place. By using shaders an interactive feedback of code changes is possible since the GPU driver compiles the source code on-the-fly. Of course, the generated geometry is downloaded to main memory and can be processed further e.g. by clustering algorithms.

5. CONCLUSIONS AND FUTURE WORK

We have presented a GPU-based framework utilizing latest features which not only allows for very fast whole brain reconstruction of white matter tracts but also enables one to implement and test fiber tracking algorithms without having to recompile C++ source code. The effect of different parameter settings on the tracking result can also be evaluated much more efficiently. In a long term this is of high clinical value since the research on fiber tracking algorithms and clinical application of fiber tracking will benefit significantly from faster implementation and higher performance of tracking algorithms.

In future work we would like to examine possibilities to filter the fibers on the GPU by user-defined ROIs. Additionally, we plan to investigate if clustering [14] and quantification [15] algorithms could benefit from GPU usage and accelerate probabilistic approaches [16]. There may also be interesting new possibilities for visualization of uncertainty using the geometry shader. Furthermore we want to parallelize the CPU algorithm to make full use of the four cores to do a more realistic comparison.

REFERENCES

[1] Reina, G., Bidmon, K., Smith, A. S., "GPU-Based Hyperstreamlines for Diffusion Tensor Imaging", IEEE Vis (2006)

[2] Petrovic, V., Fallon, J. and Kuester, F., " Visualizing Whole-Brain DTI Tractography with GPU-based Tuboids and LoD Management ", IEEE Vis (2007), Vol.13, Nr.6

[3] Peeters, T.H.J.M., Vilanova, A., ter Haar Romeny, R.B.M., "Visualization of DTI fibers using hair-rendering techniques", ASCI 2006, Lommel (2006)

[4] Hlawitschka, M., Eichelbaum, S., Scheuermann, G., "Fast and Memory Efficient GPU-based Rendering of Tensor Data", IADIS CGV 2008

[5] Mittmann, A., Comunello, E., von Wangenheim, A., "Diffusion Tensor Fiber Tracking on Graphics Processing Units", Computerized Medical Imaging and Graphics, 2008

[6] McGraw, T. and Nadar, M., " Stochastic DT-MRI Connectivity Mapping on the GPU", IEEE Vis (2007), Vol.13, Nr.6

[7] Kondratieva, P., Krüger, J. and Watermann, R., " The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization ", IEEE Vis (2005)

[8] IEEE (1985). IEEE Standard 754 for Binary Floating-Point Arithmetic, IEEE

[9] Fay, D., Sazegari, A., Connors, D., "A detailed study of the numerical accuracy of GPU-implemented math functions" In: Supercomputing 2006 workshop, 2006.

[10] MeVisLab, www.mevislab.de

[11] Geometry shader and transform feedback extensions, http://www.opengl.org/registry/

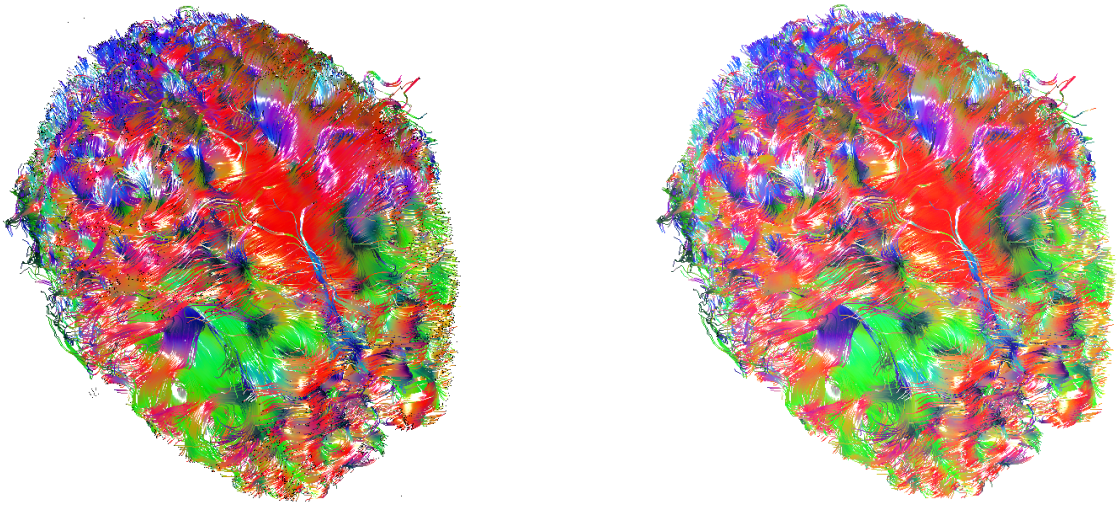
[12] Wernecke, J., "The Inventor Mentor", Addison Wesley Pub Co Inc

[13] Mori, S., Crain, B.J., Chacko, V.P., van Zijl, P.C.M., "Three dimensional tracking of axonal projection in the brain by magnetic resonance imaging", Ann. Neurol., 45, 265-269 (1999)

[14] Klein, J., Stuke, H, Stieltjes, B., Konrad, O., Hahn, H.K., Peitgen, H.-O., "Efficient Fiber Clustering using Parameterized Polynomials", SPIE Medical Imaging 2008, Vol. 6918

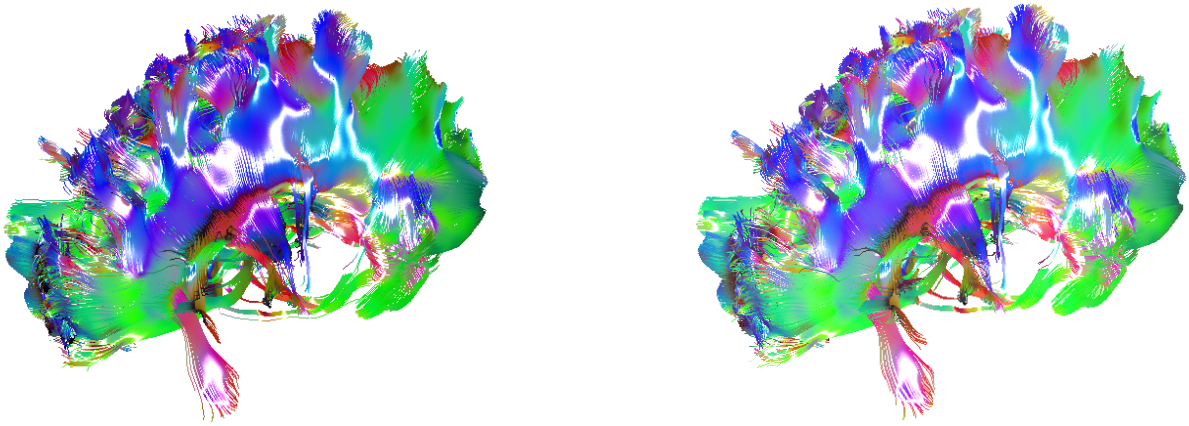
[15] Klein, J., Stuke, H., Rexilius, J., Stieltjes, B., Hahn, H.-K., Peitgen, H.-O., "Towards User-Independent DTI Quantification", SPIE Medical Imaging 2008, Vol. 6914

[16] Friman, O., Westin, C.-F., "Uncertainty in White Matter Fiber Tractography", MICCAI 2005



(a) (b)

Figure 7: Whole brain fiber reconstruction result watched from the rear left top. FA threshold was set to 0.4525. With one seed point per voxel this added up to ~30,000 points. (a) was tracked by the GPU in 118 ms (1270 ms with download and sorting of data) and (b) by CPU 1 in 5200 ms.



(a) (b)

Figure 8: Corpus Callosum tracked with 20 seed points per voxel with a total of 12,000 points. All fibers consisted of 110 points. (a) was tracked by the GPU 1 in 45 ms and (b) was tracked by CPU 1 in 2200 ms.