APPENDIX
IMPLEMENTATION DETAILS

The C++/CUDA source code is publicly available at https://git.stim.ee.uh.edu/codebase/stimlib. The majority of the source code is encapsulate in `/stim/gl/gl_spider.h`, however, data loading is handled by the `/stim/grids/image_stack.h`. Once the data is loaded using `image_stack.h`, we "attach" a `gl_spider` object to the 3D volume and read in the seed points. The source of the seed points is completely irrelevant as long as each seed contains all the necessary information described in Algorithm 1. The image stacks are loaded into GPU memory and shared with an OpenGL context using GL-Cuda inter-operability mechanic described in [1]. A `gl_spider` object can be tuned by changing the number of templates, size of the templates in pixel space and cost value. Once all of the parameters are set (or the default values are used), there are two methods in the class for finding centerpoints: `trace` and `step`. `Trace` returns all the centerpoints until termination, while `step` only executes a single step of Algorithm 1. Since each spider object is independent from all others, a user can concurrently load and attach a spider to multiple data blocks, making the implementation highly scalable. The only limitation is the amount of memory the hardware can make available, with an average number of spiders simultaneously operating being 3 for an NVidia GTX 1080 GPU. Additionally the amount of memory required is heavily dependent on the number of fibers in a volume, since the results of the segmentation are also stored in GPU memory during the segmentation process. The results are stored in a custom NTW format or wavefront obj format. The NWT format is described in the linked library. All glyphs are generated during post processing.

Our tracing method is more efficient than the traditional template matching by focusing on the voxels near the embedded network; however, a large number of volume samples are still required for the prediction (Section 3.3), position correction (Section 3.4) and radius estimation (Section 3.5) steps, which can be computationally intensive. While it is possible to manually limit the location of the sampling—a method used commonly in other vector tracking algorithms, we instead optimize the sampling, cost evaluation methods to reduce the time spent on each sample beyond the first one to a negligible value. To achieve this we take advantage of the hardware and software advances.

In the following subsections we show that performing **Predict**, **Correct**, **Fit** and **DetectBranches** and calculating cost are further optimized by performing the most time intensive operations in CUDA using shared and texture memory.

Sampling

Texture memory offers the advantage of caching the region near recent texture look-ups. This improves performance and reduces memory traffic when the read operations have some specific patterns. By storing the volume represented by the initial *image stack* as a 3D texture we take advantage of the caching mechanism in texture look-ups to make each successive sample cheaper. Each successive sample for every individual template is guaranteed to be near a sample previously taken. Each sample in V, P, R falls into a 2D texture image, with the quads resulting from S_p^a and S_p^b are placed next to each other (Fig. 1). We then specify a texture matrix transformation in order to transform the state of the tracer to texture coordinates as in Fig. 1c. The number of direction, position and size templates is constant and set during initialization, and all pixels are stored in individual textures with each array of templates allocated statically during initialization. Sampling overwrites the data stored in those arrays during each step (Figure 1).

Cost Evaluation

The cost function is evaluated on each template in V, P, R identically, an example of single instruction multiple data (SIMD) parallelism. We exploit the SIMD parallelism by using CUDA to compute kernel operations in parallel to further optimize the performance over the previous algorithm.

All arrays are stored in GPU memory until they are no longer needed, and all processing happens on the GPU, reducing the detrimental effect of the host-to-device transfer bottleneck. Each template is then processed independently and in parallel using CUDA shared memory for better performance. Each template is subtracted from the corresponding $f_{tp}(x)$ and efficiently reduced to a single value using *sequential addressing* described in [2], resulting in an unsorted array of cost values mapped one-to-one to the template array. Finally we linearly search for the value with the lowest cost and return the corresponding index. The amount of time spent searching for the minimum is small compared to the time spent on calculating the cost, therefore we chose not to optimize that aspect further.

The unrolled cylinders (see Section 4) used for branch detection are treated in a similar manner, with the exception that the memory used to store the unrolled cylinder image is allocated dynamically, since the length of the cylinder is varying. We then perform the convolution [3] with the pre-computed LoG kernel of size 5 using shared memory and padding to further accelerate the computation [4]. The cost of dynamically allocating memory is mitigated by perGB this once per fiber, as mentioned previously.

Connectivity Reconstruction

Because each new seed point generally lays close to a surface of another fiber, the network connectivity information is implicitly stored in the distance between the endpoints of single fibers and the points in all other points in the network. Of course this implication is only partially true for initial seed points. Connectivity is reconstructed as the algorithm segments each new fiber and adds it to the network. The connectivity sub-routine is called once per fiber and always at the end of segmentation; when a termination condition is reached. The termination conditions, described in Section 3.6 are closely related to the connectivity cases we handle:

1) The new fiber neither begins from a fiber, nor ends in a fiber.
2) The new fiber ends with another fiber.
3) The new fiber begins from another fiber.
4) The new fiber starts and ends with a new fiber.

Case 1 is guaranteed when we are segmenting the initial seed point. At that time, the network contains no fibers,

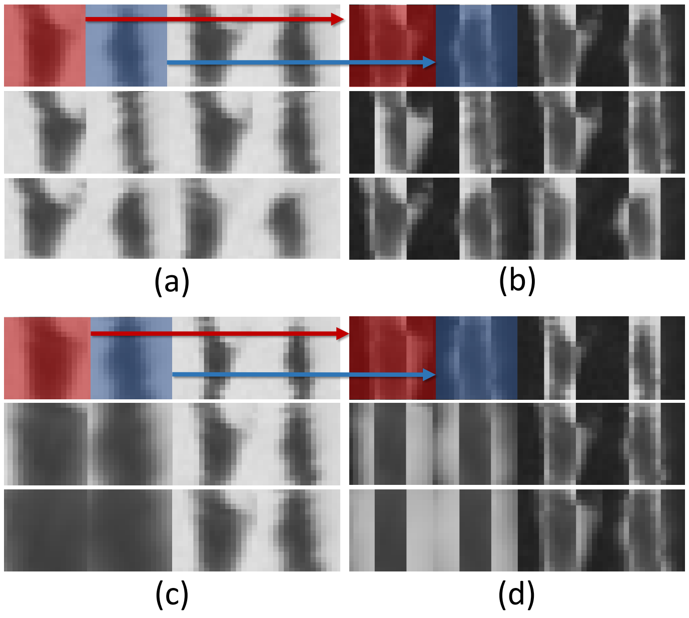

Fig. 1: A small subset of the samples $S_p^a(x, y, z)$ and $S_p^b(x, y, z)$ shown side by side, which are collected during the **Predict** (a), and **Fit**(c), respectively. The samples are then subtracted from the template and the resulting images (b,d) are reduced to a single value per sample.

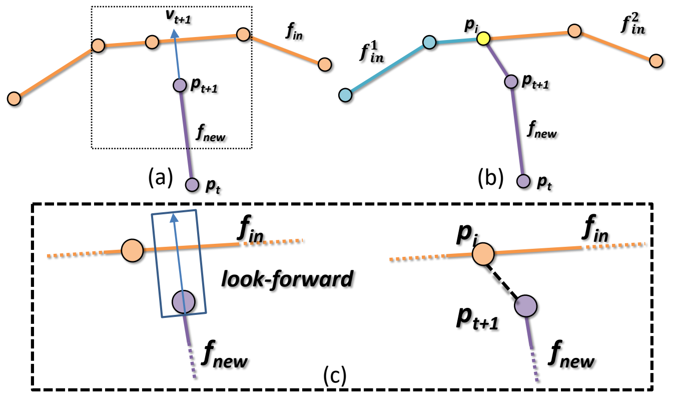

Fig. 2: After each iteration of Algorithm 1 (a) we execute a *look-forward* routine (c). We check if a fiber is detected in the limited viewport in front of the last added point $\mathbf{p}_{t+1}$ in the current fiber f_{new} (b). If a collision is detected, the subroutine finds the point $\mathbf{p}_i$ in f_{in} with the shortest distance to $\mathbf{p}_{t+1}$. The in-network fiber f_{in} is split at $\mathbf{p}_i$ into f_{in}^1 and f_{in}^2 with $\mathbf{p}_i$ remaining in all fibers including f_{new} (b). $\mathbf{p}_i$ is labeled as a node in the separately stored connectivity map.

therefore, the new fiber can neither collide, nor begin from another fiber in the network. When the network is non-empty and the new fiber is found from another initial seed point, at best we might end with another fiber; this is handled by Case 2. However, majority of the newly segmented fibers will either fall under Case 3 or Case 4.

In order to find whether we intersect another fiber previously segmented—an *in-network fiber*, we call the *look-forward* subroutine. During connectivity reconstruction, every in-network fiber is given a unique label. We test for intersections by, first, rendering the network and the new filament currently being processed. Second, we limit the area visible to the fiber along z axis such that it only detects the rendered in-network fibers if they are a distance $\delta \times r_{t+1}$ away from the point $\mathbf{p}_{t+1}$ looking in the direction $\mathbf{v}_{t+1}$. A collision happens when any fiber is detected in this small viewport. We then connect the in-network fiber f_{in} and the new fiber f_{new}. To do this we find the approximate nearest neighbor $\mathbf{p}_i$ in f_{in} to $\mathbf{p}_{t+1}$, where $\mathbf{p}_{t+1}$ is the last point in f_{new}. The two fibers are then connected at $\mathbf{p}_i$, splitting f_{in} into two fibers at $\mathbf{p}_i$. Each *look-forward* collision replaces one in-network fiber, with three new fibers: f_{in}^1,f_{in}^2, f_{new}, all containing the point $\mathbf{p}_i$. $\mathbf{p}_i$ is stored as a coordinate of a node, with connecting edges, f_{in}^1, f_{in}^2, f_{new}, for later use. The whole process is briefly illustrated in Fig. 12.

Additionally, we look at the starting point $\mathbf{p}_0$ in a similar manner but directed at $-\mathbf{v}_0$ in order to find which fiber this seed is originated from, if any, and connect those fibers using the method described above. This is the *look-back* subroutine.

References

[1] J. Neider, T. Davis, and M. Woo, "Opengl programming guide," 1993.
[2] M. Harris *et al.*, "Optimizing parallel reduction in cuda," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.
[3] B. Daga, A. Bhute, and A. Ghatol, "Implementation of parallel image processing using nvidia GPU framework," in *Advances in Computing, Communication and Control*. Springer, 2011, pp. 457–464.
[4] V. Podlozhnyuk, "Image convolution with cuda," *NVIDIA Corporation White Paper, June*, vol. 2097, no. 3, 2007.