A VTK-based, CUDA-optimized Non-Parametric Vessel Detection Method

Release 1.0

Levent Alpoge[1], Alark Joshi[2], Dustin Scheinost[3], John Onofrey[3], Xiaoning Qian[4] and Xenophon Papademetris[2,3]

January 25, 2010

[1]Half Hollow Hills High School West, Dix Hills, NY 11746-5694

[2]Department of Diagnostic Radiology, Yale University, New Haven, CT 06520-8043

[3]Department of Biomedical Engineering, Yale University, New Haven, CT 06520-8043

[4]Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620-5399

Abstract

We present a VTK-based implementation of our non-parametric vessel detection method that identifies vascular structures using a polar neighborhood profile. To accelerate the computationally intensive parts of the algorithm, we leverage the hardware capabilities in commodity graphics hardware using Compute Unified Device Architecture (CUDA). We present the results of our performance analysis and provide source code and examples to validate the reproducibility of our results.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3146]

Contents

1 Introduction

In the field of cardiovascular pathology, the existence of a highly accurate, automated vessel detection method is paramount. With current vascular segmentation algorithms, there is an inherent assumption that each voxel belongs to a single cylinder. These techniques, which are based on identifying a single cylindrical structure perform particularly well for long, structured cylindrical regions in vascular data. Indeed, the large majority of techniques, including the matched filters of [2] and [8], and the Hessian-tensor based [3] and [7] are particularly good at detecting vessel center points but fall short of single-scale comprehensive vascular segmentation. While multiscale implementations of these methods do end up sufficing in the majority of cases for the purposes of basic angiography, a much more robust and accurate method that consistently detects branching points and vascular extents is needed for higher-level vascular structural analysis (e.g., vascular tree reconstruction).

We present the results of a VTK-based implementation of a non-parametric vascular detection technique described in [6]. The polar neighborhood intensity profile underlies this vessel detection algorithm, with the key intuition that vascular points are both globally bright and have at least one local orientation with low intensity deviation. Our implementation of this algorithm follows the standard algorithmic filter model in VTK in its modules, and leverages hardware accelerated versions of FFT and RFFT in CUDA ([1], [5]).

The input image and other parameters are read from the command line by a Tcl script, which invokes the VTK-based source code to perform vascular detection. The convolution step can be computationally expensive with filter kernels getting quite big (70x70x70). To accelerate this process, we compute the FFT of the image as well as the kernel, multiply the two images and then take the inverse FFT of the resulting image. The convolution step is even more accelerated by using the hardware accelerated version of FFT/RFFT in CUDA.

The rest of the paper is structured as follows: In Section 2, a brief overview of our entropy-based vesselness measure is discussed. In Section 3, our implementation and optimizations are presented. In Section 4, experimental results are reported. The final section of this paper consists of a discussion of results and a conclusion of this work.

2 The Entropy-based Vesselness Measure

In the non-parametric vessel detection method [6], rather than assuming that a voxel belongs to at most one oriented structure, it is observed that a voxel belonging to a vascular structure tends to be both globally bright as well as locally comparable in at least one direction to its neighbors–in the sense that, considering a spherical neighborhood about the voxel, there is at least one "wedge" of the spherical neighborhood that consists of relatively proximal voxels in brightness (identified by deviation in the wedge). Figure 1 depicts partitioning of the neighborhood in the 2D and 3D case.

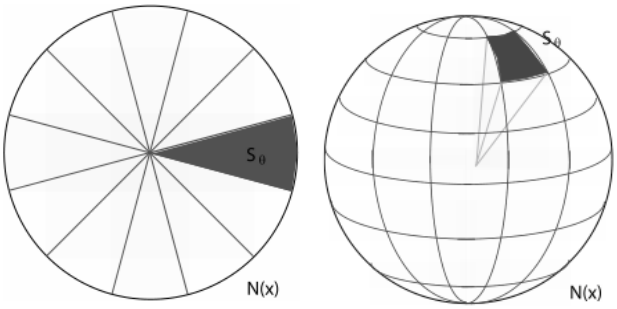

Figure 1: Partitioning the neighborhood into wedges. On the left is a 2D example, where a circular neighborhood is partitioned into 2D wedges whereas on the right is a 3D example showing the partition of a spherical neighborhood into 3D wedges.

2.1 The Polar Neighborhood Intensity Profile

Let us formalize this intuition by beginning with our observation that vascular points are locally close to their neighbors in brightness in certain directions. Consider the spherical neighborhood about a given voxel. On dividing this neighborhood into wedges that span the full sphere, we compute the average squared intensity deviation over all relative directions $\theta = \{\psi, \phi\}$ (with $\psi \in [0, 2\pi)$, $\phi \in [0, \pi)$) for each voxel $\mathbf{x}$ as:

$$Dev(\mathbf{x}, \theta) = \int h(\mathbf{u})(I(\mathbf{x}-\mathbf{u}) - I(\mathbf{x}))^2 \, d\mathbf{u} \,, \tag{1}$$

where I is the intensity of the image, and $h(\mathbf{u})$ is a function of relative position to $\mathbf{x}$. h essentially acts as a partition characteristic function of the spherical neighborhood about $\mathbf{x}$, and different $h(u)$ gives different partition functions or weighting schemes to collect statistics of brightness in the neighborhood. Here x, y, z refer to the x, y, z-coordinates of the voxel $\mathbf{x}$ under consideration.

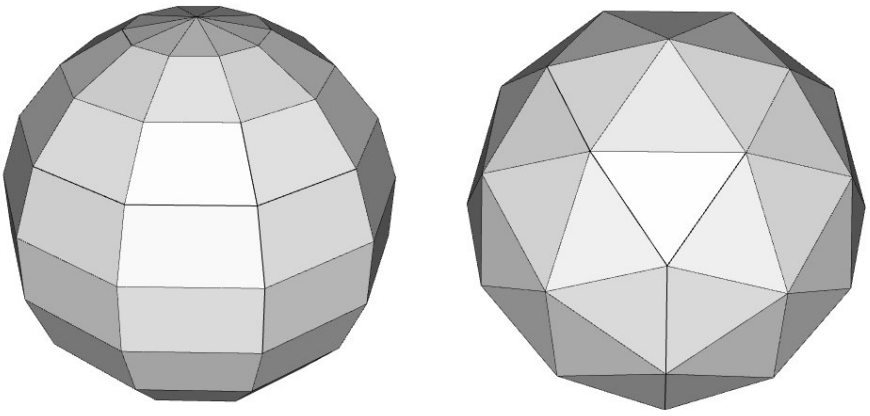

Figure 2: Our wedging implementations. The left image shows the spherical neighborhood sampling method which results in non-uniform sampling at the poles. The right image shows icosahedron-based sampling which results in uniform sampling over the neighborhood.

We implemented two different partitioning schemes for the spherical neighborhood.

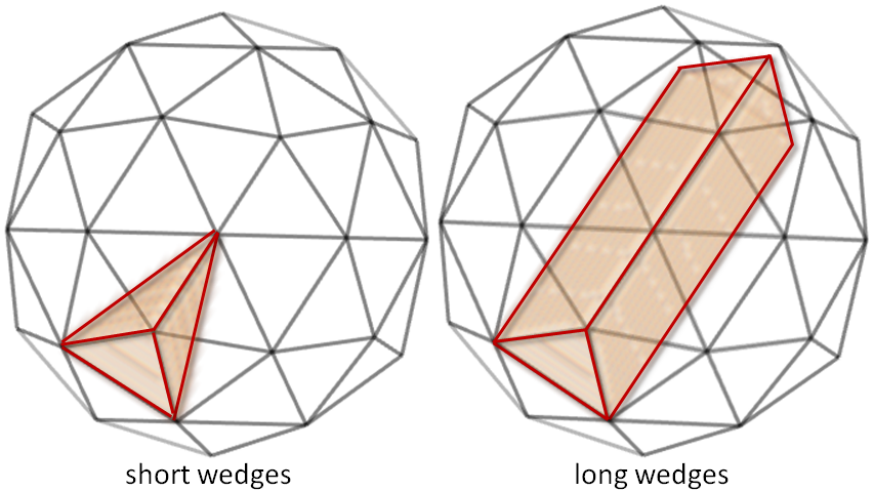

Figure 3: Wedge-based distributions of the subdivided icosahedron. The left image shows a short wedge which converges at the center. The right image shows a long wedge where the triangular prism-shaped wedge connects triangles on opposing faces of the sphere.

1. Spherical separable filter $h(\mathbf{u}) = h_r(\mathbf{u})h_\theta(\mathbf{u}) = h_r(r)h_\theta(\psi,\phi)$, with $r = \sqrt{x^2+y^2+z^2}$, $\psi = \arctan(y/x)$, and $\phi = \arctan(z/\sqrt{x^2+y^2})$, which gives $n_\psi \times n_\phi$ wedges, distributed as shown in left image in Figure 2. This causes non-uniform sampling over the spherical neighborhood due to over-sampling at the poles.

2. To address the non-uniform sampling of the spherical neighborhood, we have implemented a way to obtain uniform sampling by subdividing an icosahedron. The uniform icosahedron-based sampling can be seen in the right image in Figure 2. We subdivide the icosahedron in two ways which leads to "short" and "long" wedges, as shown in Figure 3. For "short" wedges, each triangle $T = (\mathbf{x},\mathbf{y},\mathbf{z})$ is connected to the center point $\mathbf{c}$, creating a tetrahedral wedging of the icosahedron. For "long" wedges, each triangle $T = (\mathbf{x},\mathbf{y},\mathbf{z})$ is connected to its "opposite" triangle, forming a triangular prism. Note that the "opposite" face of an icosahedron is actually the original triangle reflected through the center, $\mathbf{c}$. Hence, a rotation by $\frac{\pi}{2}$ (or reflection through its own centroid) must be done to align the triangles. This construction can be obtained quickly through coordinate geometry per the connection of T and T', which is defined as follows:

$$T' = \left(2\mathbf{c} - \frac{2\mathbf{y}+2\mathbf{z}-\mathbf{x}}{3}, 2\mathbf{c} - \frac{2\mathbf{x}+2\mathbf{z}-\mathbf{y}}{3}, 2\mathbf{c} - \frac{2\mathbf{x}+2\mathbf{y}-\mathbf{z}}{3}\right)$$

We continue with our formulation by expressing the probability of our voxel $\mathbf{x}$ having, in the orientation θ, small intensity variation, $p_v(\mathbf{x},\theta)$ as:

$$p_v(\mathbf{x},\theta) = ce^{-\beta Dev(\mathbf{x},\theta)}, \tag{2}$$

where c is the normalization factor and β is the emphasizing factor.

We notice that this gives us a probability density function describing local brightness over our image, and recall that the entropy, H, of a density function describes its spread. Therefore, we may compute the "tightness measure" $v(\mathbf{x})$ as:

$$v(\mathbf{x}) = e^{-\tau H(p_v)} = e^{\tau \int p_v(\mathbf{x},\theta)\log p_v(\mathbf{x},\theta)\,d\theta}. \tag{3}$$

2.2 Local Brightness

We continue to the second part of our observation: candidate voxels are globally bright. To formalize this notion, we define a brightness function $b(\mathbf{x})$:

$$b(\mathbf{x}) = s(\mu(I_{S_{\theta_{min}}}(\mathbf{x})) - \mu(I_{S_{\theta_{max}}}(\mathbf{x}))), \tag{4}$$

where $s(k) = 1/(1+e^{-\alpha k})$ is a sigmoid function, $\mu(I_{S_{\theta_{min}}}(\mathbf{x}))$ is the average intensity of the wedge with minimum deviation, and $\mu(I_{S_{\theta_{max}}}(\mathbf{x}))$ is the average intensity of the wedge with maximum deviation.

In our implementation, however, we use a simpler brightness measure. Observe that for a voxel to be labeled globally bright, its intensity must be large relative to the range of values taken on by the rest of the image. Since we implement this algorithm by first normalizing our input image, we smooth the input image using `vtkImageGaussianSmooth` and take the intensity value at each smoothed voxel to be that voxel's "brightness measure".

2.3 The Polar Profile Vesselness Measure

Our entropy-based polar profile vesselness measure $PPV(\mathbf{x})$ is defined as the product between the "tightness" and "brightness" measures $v(\mathbf{x})$ and $b(\mathbf{x})$:

$$PPV(\mathbf{x}) = b(\mathbf{x}) \times v(\mathbf{x}). \tag{5}$$

3 Implementation and Optimizations

3.1 Implementation

We implement the algorithm within the new framework of the VTK pipeline. [1]. Our main image-to-image filter is written in Tcl and uses classes written in C++, derived from appropriate VTK parent classes. The code execution is described below as well as in Figure 4.

1. We first load the input image (in `.vtk` format) into a convolution filter, `vtkNewImageConvolution` to compute, for each voxel $\mathbf{x}$, $p_v(\mathbf{x},\theta)$ for each θ, with optional outputs giving $Dev(\mathbf{x},\theta)$ and $\mu(I_{S_\theta}(\mathbf{x}))$(e.g., for the purposes of computing the original "brightness measure").

2. This filter constructs neighborhoods based on user input, with the values of `usenewsphere` (use the icosahedral neighborhoods) and `usenewwedges` (use the "long wedges", as described in the previous section) options specifying the precise constructions.

3. If `usenewsphere` is 1, `vtkjoColoredSphereSource`, which derives from `vtkjoSphereSource`, is called to construct these neighborhoods (based on `usenewwedges`), whereas otherwise the neighborhoods and wedges are constructed inside `vtkNewImageConvolution` itself.

4. We then pass this information into `vtkNewODFVesselnessFilter` to compute the entropy and "tightness measure", and produce the resultant "vesselness" image, using `vtkImageMathematics`, with our Gaussian-smoothed input to arrive at the resultant Polar Profile Vesselness output.

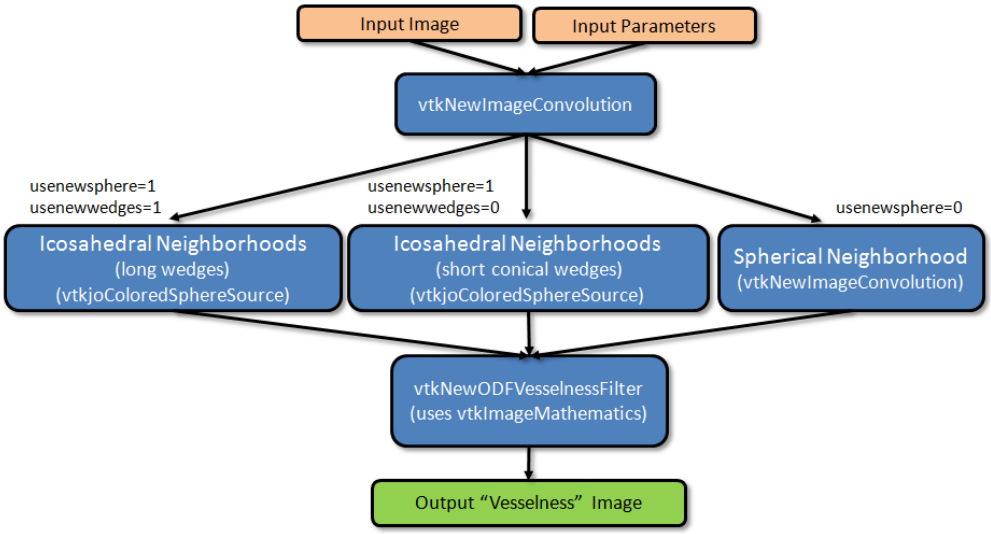

Figure 4: This diagram shows the way in which the input image is processed based on the selected parameters to give an output "vesselness" image. Here we show the three ways in which the neighborhood around a voxel can be divided. The first two methods are the result of subdividing an icosahedron (`vtkjoColoredSphereSource`), while the third way uses a spherical neighborhood-based sampling (`vtkNewImageConvolution`).

3.2 Optimizations

Originally, this implementation took about 4 hours for a 100 x 256 x 100 image on a 2.0 GHz Intel Xeon CPU running Windows XP [4]. Clearly, for a large database of images, this is a critical bottleneck for analysis. To significantly speed up the running time of our code, we implemented its most computationally expensive procedure–the calculation of $Dev(\mathbf{x},\theta)$–on a Graphics Processing Unit (GPU). This was done using the capabilities in CUDA (hardware acceleration library for programming graphics processing units). Recall from equation (1) that,

$$Dev(\mathbf{x},\theta) = \int h(\mathbf{u})(I(\mathbf{x}-\mathbf{u})-I(\mathbf{x}))^2\,d\mathbf{u}\,.$$

Therefore, expanding out the integrand:

$$\begin{aligned} Dev(\mathbf{x},\theta) &= \int h(\mathbf{u})(I(\mathbf{x}-\mathbf{u}))^2 - 2(I(\mathbf{x}-\mathbf{u})I(\mathbf{x})) + (I(\mathbf{x})^2)\,d\mathbf{u} \\ &= \int h(\mathbf{u})I(\mathbf{x}-\mathbf{u})^2\,d\mathbf{u} - \int 2h(\mathbf{u})(I(\mathbf{x}-\mathbf{u})I(\mathbf{x}))\,d\mathbf{u} + \int h(\mathbf{u})I(\mathbf{x})^2\,d\mathbf{u} \\ &= (h*I^2)(\mathbf{x}) - 2I(\mathbf{x})((h*I)(\mathbf{x})) + I(\mathbf{x})^2 \int h(\mathbf{u})\,d\mathbf{u} \end{aligned}$$

Note that if we normalize the wedges such that $\int h(\mathbf{u})\,d\mathbf{u} = 1$ we obtain:

$$Dev(\mathbf{x},\theta) = (h*I^2)(\mathbf{x}) - 2I(\mathbf{x})((h*I)(\mathbf{x})) + I(\mathbf{x})^2\,.$$

Recalling the Convolution Theorem, which states that, for functions f and g: $(f*g) = \mathcal{F}^{-1}\{\mathcal{F}\{f\}\times\mathcal{F}\{g\}\}$, we can change this into a more usable form:

$$Dev(\mathbf{x},\theta) = (\mathcal{F}^{-1}\{\mathcal{F}\{h\}\times\mathcal{F}\{I^2\}\})(\mathbf{x}) - 2I(\mathbf{x})((\mathcal{F}^{-1}\{\mathcal{F}\{h\}\times\mathcal{F}\{I\}\})(\mathbf{x})) + I(\mathbf{x})^2 \qquad (6)$$

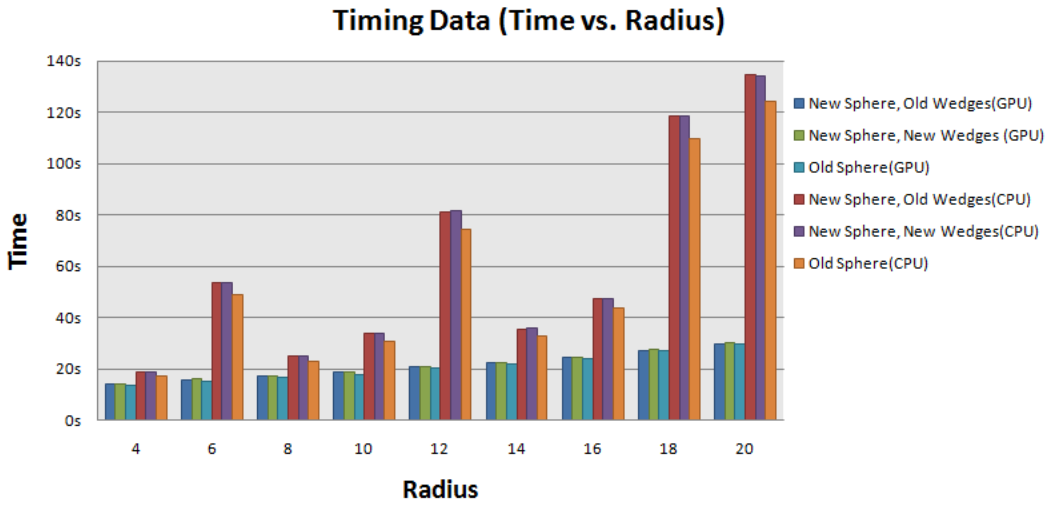

Figure 5: Timing results for the various implementations of the algorithm. The GPU versions of all the algorithms are significantly faster regardless of the neighborhood radius used.

It is well-known that the Fast Fourier Transform (FFT) and its inverse (RFFT) are highly parallelizable. We leverage the optimized FFT/RFFT functionality [5] in CUDA to accelerate our convolution step. We wrote the classes `vtkNewCUDAFFT` and `vtkNewCUDARFFT`, as well as the function and kernel inside of `NewCUDAFFT.cu` and `NewCUDAFFT_kernel.cu`, which invokes CUDA's FFT/RFFT functionality. Our measured speedup of approximately 18 times (470 times relative to the original implementation) considerably speeds up the execution time on a 100 x 256 x 100 image, as shown in Figure 5. Note that the GPU implementations of all the versions are significantly faster than their CPU versions and scale well for larger radii. Table 1 provides the timing results for each variation of the algorithm.

Our optimization is coded as follows:

Listing 1: ../Source/vtkNewImageConvolution.cpp

```
1  //
       ------------------------------------------------------------------------------------------
2  void vtkNewImageConvolution::SimpleExecute(vtkImageData* input,vtkImageData* output)
3
4    vtkImageData *FI=NULL, *FI2=NULL;
5      {
6      if (out2!=NULL)
7        out2->FillComponent(frame,0.0);
8          gettimeofday(&start, NULL);
9    starttime=start.tv_sec+(start.tv_usec/1000000.0);
10 #endif
11           endtime=end.tv_sec+(end.tv_usec/1000000.0);
12         }
13
14 #endif
15       for (int j=0;j<nt;j++)//compute Dev per our previously derived formula
16           if (j==debugv && this->Debug)
17       }
18           fprintf(stderr,"Mode=0, frame=%d\n",frame);
19     out->GetSpacing(spa);
20   }
```

Timing results for the various implementations (seconds)						
Radius	New Sphere Old Wedges (GPU)	New Sphere New Wedges (GPU)	Old Sphere GPU version	New Sphere Old Wedges (CPU)	New Sphere New Wedges (CPU)	Old Sphere CPU version
4	14.06	14.00	13.55	18.88	18.84	17.19
6	15.77	16.07	15.24	53.59	53.42	48.65
8	17.38	17.47	16.49	25.21	25.18	22.99
10	18.89	18.80	17.96	33.74	33.81	30.80
12	21.00	21.00	20.14	81.24	81.55	74.42
14	22.42	22.38	21.63	35.44	35.66	32.84
16	24.45	24.50	23.75	47.43	47.54	43.94
18	27.24	27.37	26.83	118.47	118.51	109.69
20	29.70	30.27	29.85	134.73	133.90	124.12

Table 1: This table shows the time required for the various implementations of the algorithm. Notably, the GPU versions are faster for all the radii and, more importantly, provide a significant speedup for higher values of the spherical neighborhood radius.

```
21        comp->SetInput(img);
22        tmp=this->PadImage(comp->GetOutput(),paddim);
23      }
24      }
25          fprintf(stderr,"\nUsing CUDA\n");
26    vtkNewCUDAFFT* fft2=vtkNewCUDAFFT::New();//implement CUDA's FFT/RFFT functionality
27        fft2->SetDebugMode(this->Debug);
28    fft2->SetForward(1);//set to FFT
29        fft2->SetTimingMode(this->TimingMode);
30        fft2->SetDoComplex(this->DoComplexFFTs);
31        //      if (inData!=NULL)
32        //  fft2->SetinData(inData);
33    fft=fft2;
34      }
35    else
36      }
37    if (dosquare)
38      {
39    double range1[2];
40    if (this->Debug)
41    double range2[2];
42    if (this->Debug)
43    if (this->Debug)
44        dim1[0],dim1[1],dim1[2],nc1,
45        dim2[0],dim2[1],dim2[2],nc2,
46        ori1[0],ori1[1],ori1[2],
47    math->SetOperationToComplexMultiply();
48    if (usegpu)
49
50    }
51  #endif
```

Listing 2: ../Source/vtkNewCUDAFFT.cpp

```
1  extern "C" void NewCUDAFFT(float *input, int dim[3], int forward, int doComplex, int
       time);
2  void vtkNewCUDAFFT::SimpleExecute(vtkImageData* input ,vtkImageData* output)
```

```
{
  NewCUDAFFT(inputdata, dim, this->Forward, this->DoComplex, this->TimingMode);
```

Listing 3: ../Source/NewCUDAFFT.cu

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <cufft.h>
extern "C" void NewCUDAFFT(float *input, int dim[3], int forward, int doComplex, int
    time)
{
    cufftComplex *d_in;
    cufftComplex *d_out;
    cufftComplex *h_in;
    cufftComplex *h_out;
    int x = dim[0];
    int y = dim[1];
    int z = dim[2];
    int size=x*y*z;
    int mem_size=sizeof(cufftComplex)*size;
    h_in=(cufftComplex*)malloc(mem_size);
    h_out=(cufftComplex*)malloc(mem_size);
    for(int i = 0; i < size; i++)//initialize values
      {
      h_in[i].x = input[2*i];
      h_in[i].y = input[2*i+1];
      }
    cudaMalloc((void**)&d_in, mem_size);//allocate memory on device
    cudaMemcpy(d_in, h_in, mem_size, cudaMemcpyHostToDevice);//copy memory to device
    cufftHandle planForward;//create a plan--much like LAPACK's procedure for
        optimization
    cufftPlan3d(&planForward, x, y, z, CUFFT_C2C);//initialize plan
    d_out=d_in;
    if (forward==1)
      cufftExecC2C(planForward, d_in, d_out, CUFFT_FORWARD);//execute FFT
    else
      cufftExecC2C(planForward, d_in, d_out, CUFFT_INVERSE);//execute RFFT
    cudaMemcpy(h_out, d_out, mem_size, cudaMemcpyDeviceToHost);
    int normalizer;//note: when using CUFFT, upon RFFT a normalization factor of 1/
        pixels = (pixels^(-1/2))^2 must be applied, since the device doesn't normalize
        on its own
    if(forward == 0)
      normalizer = size;
    else
      normalizer = 1;
    for(int i=0;i<size;i++)//copy out results
      {
      input[i*2] = h_out[i].x/normalizer;
      input[(i*2)+1] = h_out[i].y/normalizer;
      }
}
```

4 Results

Our implementation was tested on a synthetic 80 x 80 x 80 image consisting of 3 cylinders, a sphere, and a central cylinder with a branching point, as shown in Figure 6. In the 3D representation, the left image shows

the original synthetic input image and the right image shows the output of our algorithm. Various structures such as branch points, cylindrical and spherical structures are identified by our algorithm.

To highlight certain features of our algorithm, zoomed in representations are shown in Figure 7. The left set (A), shows the branch points highlighted by the red circle, where the left image is the original synthetic image and the right image shows the output of our algorithm. The branch point and connecting cylindrical structures are accurately detected. In the right set (B) of Figure 7, we highlight the ability of our algorithm to perform equally well on cylindrical as well as spherical features. Spherical structures are crucial, since aneurysms in a vascular data need to be identified and assuming that all the structures are cylindrical in such dataset limits the identification of such structures.

The input image (`syntvessel.vt`), output image (`syntvessel_vesselness.vt`), and Tcl script (`vesselness.tcl`) used to run the algorithm are provided in the submission. The command used to execute the algorithm in this case was `vtk vesselness.tcl syntvessel.vt. vesselness.tcl` takes optional inputs of:

- β - the emphasizing factor (default value 800.0)
- r - the radius of the neighborhood constructed (default value 8)
- `usenewsphere` - the command line switch specifying whether to use spherical or icosahedral neighborhoods (default value 0, i.e. spherical)
- `usenewwedges` - the command line switch specifying whether to use "short" or "long" wedges (as specified in the Implementation section, default value 0, i.e. "short" wedges)
- `forceCPU` - the command line switch specifying whether to use CUDA and execute code on the GPU (if it recognizes CUDA) or execute fully on the CPU (default value 0, i.e. use GPU)
- `outstem` - the stem of the output (default value of the stem of the input file)
- `debug` - the command line switch specifying whether or not to print debug statements (default value 0, i.e. do not print).

5 Conclusion

We present an implementation of the non-parametric vessel detection method [6] in the framework of the Visualization Toolkit's filter pipeline, with significant acceleration-based optimizations using CUDA. The implementation was found to achieve a speedup of approximately 18 times for a 100 x 256 x 100 image (about 470 times faster than the original implementation) and gives strong results for synthetic vessel data.

6 Acknowledgments

We would like to thank Dr. Themis Kyriakides and Chrysi Notskas for making this collaboration possible. This work was supported in part by the NIH/NIBIB under grant R01 EB006494 (Papademetris, X. PI). All the code that accompanies this paper is made available through the same license as VTK. We acknowledge that some of this code derives from original VTK code.

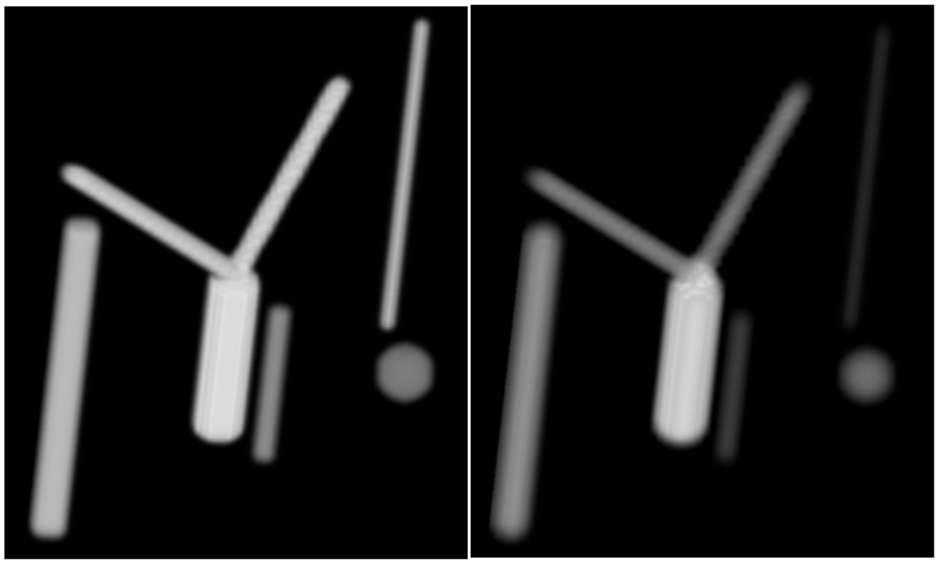

Figure 6: Three dimensional visualization of the synthetic test image and the vesselness output. Branch points, spherical and cylindrical structures can be identified using our algorithm.

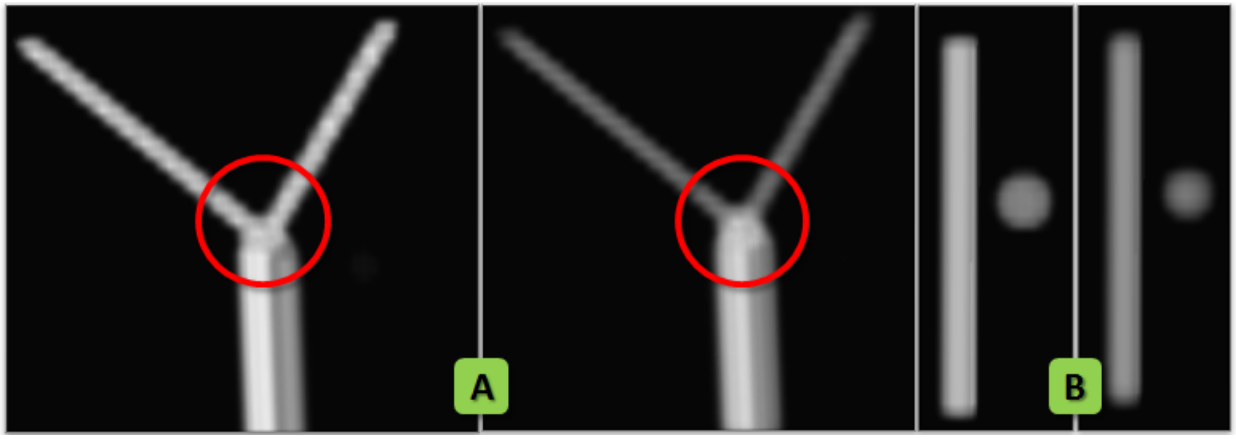

Figure 7: Image Set (A) shows a closer look at the branch point detection ability of our algorithm. As shown by the red circle, our algorithm is able to detect branch points. Similarly, Image Set (B) shows the ability of our algorithm to identify cylindrical and spherical structures which is crucial for the vascular datasets where there may be an aneurysm.

References

[1] Generalized vtk data processing framework. Technical report, Kitware Inc., 2003. 1, 3.1

[2] R. N. Czerwinski, D. L. Jones, and W. D. O'Brien Jr. Line and boundary detection in speckle images. *IEEE Transactions on Image Processing*, 7:1700–1714, 1998. 1

[3] A. F. Frangi, W. J. Niessen, K. L. Vincken, and M. A. Viergever. Multiscale vessel enhancement filtering. *MICCAI, Lecture Notes in Computer Science*, 1496:130–137, 1998. 1

[4] A. Joshi, X. Qian, D. P. Dione, K. R. Bulsara, C. K. Breuer, A. J. Sinusas, and X. Papademetris. Effective visualization of complex vascular structures using a non-parametric vessel detection method. *IEEE Transactions on Visualization and Computer Graphics*, 14:1603–1610, 2008. 3.2

[5] nVidia. *CUFFT Library*, 2007. 1, 3.2

[6] X. Qian, M. Brennan, D. Dione, L. Dobrucki, M. Jackowski, C. Breuer, A. Sinusas, and X. Papademetris. A non-parametric vessel detection method for complex vascular structures. *Medical Image Analysis*, 13:49–61, 2008. 1, 2, 5

[7] Y. Sato, S. Nakajima, N. Shiraga, H. Atsumi, T. Koller, G. Gerig, and R. Kikinis. Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images. *Medical Image Analysis*, 2:143–168, 1998. 1

[8] M. Sofka and C. V. Stewart. Retinal vessel centerline extraction using multiscale matched filters, confidence and edge measures. *IEEE Transactions on Medical Imaging*, 25:1531–1546, 2006. 1