Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Erik Smistad · Frank Lindseth

Multigrid gradient vector flow computation on the GPU

the date of receipt and acceptance should be inserted later

Abstract Gradient vector flow (GVF) is a feature-preserving spatial diffusion of image gradients. It was introduced to overcome the limited capture range in traditional active contour segmentation. However, the original iterative solver for GVF, using Euler's method, converges very slowly. Thus many iterations are needed to achieve the desired capture range. Several groups have investigated the use of graphic processing units (GPUs) to accelerate the GVF computation. Still, this does not reduce the number of iterations needed. Multigrid methods, on the other hand, have been shown to provide a much better capture range using considerable less iterations. However, non-GPU implementations of the multigrid method are not as fast as the Euler method when executed on the GPU. In this paper, a novel GPU implementation of a multigrid solver for GVF written in OpenCL is presented. The results show that this implementation converges and provides a better capture range about 2-5 times faster than the conventional iterative GVF solver on the GPU.

Keywords Gradient Vector Flow · GPU · Multigrid

1 Introduction

Gradient vector flow (GVF) is a feature-preserving spatial diffusion of image gradients. The GVF field is defined as the vector field **V**, that minimizes the energy function E:

$$E(\mathbf{V}) = \int \mu |\nabla \mathbf{V}(\mathbf{x})|^2 + |\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x})|^2 |\mathbf{V}_0(\mathbf{x})|^2 d\mathbf{x} \quad (1)$$

where $\mathbf{V}_0$ is the initial vector field. The first part of this integrand $|\nabla \mathbf{V}(\mathbf{x})|$, is the diffusion part that favors a vector field that is smooth. The second part $|\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x})|$, on the other hand, is the feature-preserving part that pushes the vector field to be similar to the initial vector field. The last part $|\mathbf{V}_0(\mathbf{x})|$ reduces the feature preservation for weak edges so that they are smoothed out instead. The parameter μ governs how much the vector field should be smoothed. Thus μ should be increased if there is a lot of noise. Also, note that the gradient operator ∇ is applied separately for each vector component.

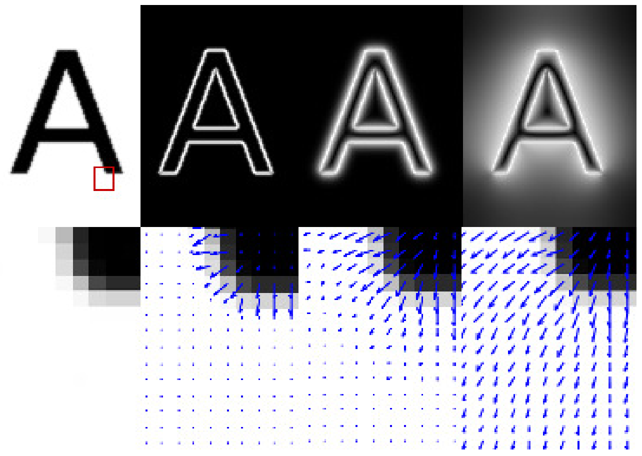
Fig. 1: Example of GVF execution using Euler's method. The image to the left is the input image and the three next images show the GVF vector field after 0, 10 and 400 iterations. The top row shows the magnitude of the vector field and the bottom row shows the vectors superimposed on a zoomed area of the input image.

Figure 1 depicts the process of the GVF algorithm. The image to the left is the input image. Next, is the

Erik Smistad · Frank Lindseth
Dept. of Computer and Information Science
Norwegian University of Science and Technology
Sem Saelandsvei 7-9, NO-7491 Trondheim
Tlf.: +47 73594475
E-mail: smistad@idi.ntnu.no

Erik Smistad · Frank Lindseth
SINTEF Medical Technology

initial vector field $\mathbf{V}_0$ and the next two images show the vector field $\mathbf{V}$ after 10 and 400 iterations. The top row shows the magnitude of the vectors fields while the bottom row shows the vectors superimposed on a zoomed area of the input image. The initial image shown top-left is an image smoothed by convolution with a Gaussian.

The GVF algorithm was introduced by Xu and Prince [19] as a new external force field for active contours (AC). Also known as snakes or deformable models, AC are curves that move in an image while trying to minimize their energy and are used extensively for boundary detection and segmentation. The original snake, introduced by Kass et al. [12], has the problem of getting stuck in boundary concavities and low capture range. The capture range is how far from the object's border a snake can be initialized and still converge to the border. The GVF method is able to overcome both these problems.

After its introduction, the GVF algorithm has been applied for several other image processing applications. Bauer and Bischof [3] developed a novel approach to use the GVF as a replacement for the scale-space framework in Hessian based tube detection. Hassouna and Farag [10] and Bauer and Bischof [4] used the GVF to extract skeletons from objects. Ray and Acton [14] used GVF to track leukocytes from intravital video microscopy. Guo and Lu [8] argued that GVF combined with mutual information can improve multi-modal image registration.

Xu and Prince [19] showed that the GVF field can be found by solving the Euler equation:

$$\mu\nabla^2\mathbf{V}(\mathbf{x}) - (\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x}))|\mathbf{V}_0(\mathbf{x})|^2 = \mathbf{0} \tag{2}$$

This can be done by treating the vector field $\mathbf{V}$ as a function of time and using Euler's method:

$$\begin{aligned}\mathbf{V}(\mathbf{x}, t+1) =&\mathbf{V}(\mathbf{x}, t) + \mu\nabla^2\mathbf{V}(\mathbf{x}, t) - \\ &(\mathbf{V}(\mathbf{x}, t) - \mathbf{V}(\mathbf{x}, 0))|\mathbf{V}(\mathbf{x}, 0)|^2\end{aligned} \tag{3}$$

Algorithm 1 shows how this is done numerically.

Algorithm 1 3D Gradient vector flow using Euler's method

```
Input: Initial vector field $\mathbf{V}_0$ and the constant $\mu$.
$\mathbf{V} \leftarrow \mathbf{V}_0$
for a number of iterations do
    for all voxels $\mathbf{x}$ do
        $L \leftarrow -6\mathbf{V}(\mathbf{x}) + \mathbf{V}(x+1, y, z) + \mathbf{V}(x-1, y, z) + \mathbf{V}(x, y+1, z) + \mathbf{V}(x, y-1, z) + \mathbf{V}(x, y, z+1) + \mathbf{V}(x, y, z-1)$
        $\mathbf{V}_n(\mathbf{x}) \leftarrow \mathbf{V}(\mathbf{x}) + \mu L - (\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x}))|\mathbf{V}_0(\mathbf{x})|^2$
    end for
    $\mathbf{V} \leftarrow \mathbf{V}_n$
end for
```

Calculating the GVF field serially using this numerical approach is slow due to the need for many iterations to converge. However, since each pixel is calculated independently of the other pixels, each pixel can be processed in parallel with the same instructions for each iteration. This data parallelism makes the GVF ideal for execution on graphic processing units (GPUs). The GPU is a type of single instruction, multiple data (SIMD) processor. It can perform the same instruction on each element in a dataset in parallel. GPUs achieve this with many functional units (e.g. ALUs) that share control units.

Because of the simplicity and data parallelism of Euler's method for solving GVF (see Algorithm 1), there exist several GPU implementations of this method. Eidheim et al. [6], He and Kuester [11] and Zheng and Zhang [20] all presented GPU implementations of GVF and active contours for 2D images using shader languages. A GPU implementation of 2D GVF written in CUDA was done by Alvarado et al. [2]. In our previous work [16], we presented a highly optimized GPU implementation of GVF for both 2D and 3D images using OpenCL. This implementation uses both texture memory and a 16-bit storage format to reduce memory latency and has been used for fast segmentation and centerline extraction of tubular structures in medical images [15,17]

Han et al. [9] proposed an alternative numerical scheme to Euler's method using a multigrid method. Their results showed significant improvement in speed and quality.

There exist several implementations of multigrid methods on the GPU. Some examples are Bolz et al. [5] who implemented a sparse matrix multigrid solver on the GPU and Grossauer and Thomas [7] who implemented a denoising filter and a solver for optical flow on the GPU using multigrid methods. However, to our knowledge, there are no published implementations on multigrid methods for GVF on the GPU.

In this paper, we present a parallel GPU implementation of GVF for 3D images using the numerical multigrid scheme presented by Han et al. [9]. The implementation is available online.

The next section describes the multigrid solver for GVF and how it was implemented and optimized for the GPU. The implementation was evaluated on several large medical 3D datasets and execution time and average error are reported in the result section. Finally, a discussion of the results and conclusions are presented.

2 Methods

2.1 Multigrid gradient vector flow

Throughout this article, a *computational grid* refers to the current vector field $\mathbf{V}$ with a specific resolution. While Euler's method only work on one computational grid with one specific resolution, multigrid (MG) solvers work on several computational grids with different resolutions. Thus MG methods are a type of multiresolution methods. The general idea of MG methods is to accelerate

http://github.com/smistad/GPU-Multigrid-Gradient-Vector-Flow/

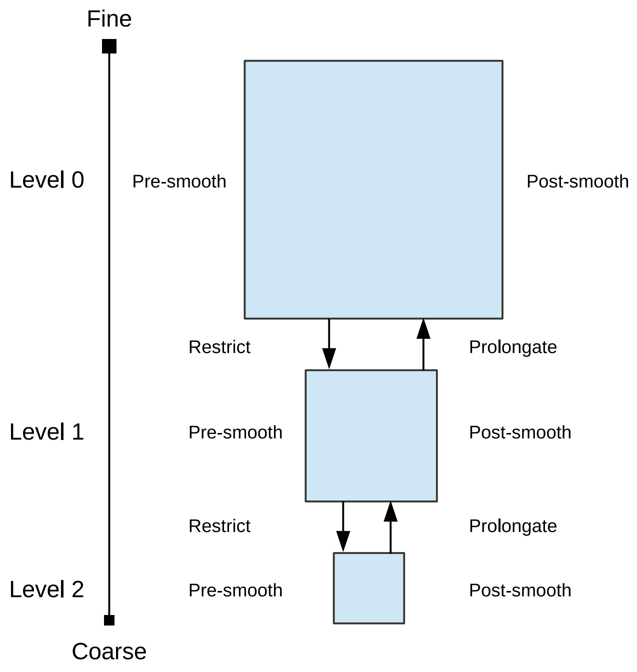

Fig. 2: The multigrid V-cycle with 1 levels.

the convergence by solving the same problem only on a coarser computational grid and then use this solution when solving the finer grid. Thus this is a recursive method and for each recursive call there are five steps:

1. Pre-smoothing: Smooth the current grid to remove high frequency errors.
2. Restriction: Create a coarser grid of the current grid.
3. Run this method recursively on the coarser grid from the previous step.
4. Correction: Prolongate/Interpolate the solution of the previous step to the same resolution as the current grid and use it to correct the current solution.
5. Post-smoothing: Smooth the current grid again.

This is called the V-cycle and is depicted in Figure 2. In the next sections, these steps are explained in more detail. Note that for each of these steps there are several choices of methods and parameters. Han et al. [9] investigated which of these choices gave the best convergence rate for GVF. Thus in this study, the same methods and parameters have been used.

2.1.1 Smoothing

The purpose of the pre- and post-smoothing is to reduce high frequency errors. This is done using the red-black Gauss-Seidel (RBGS) relaxation method. The advantage of using the RBGS method versus the default lexiographic Gauss-Seidel method is that RBGS allows half of the voxels in the grid to be computed in parallel. Which is crucial for the GPU implementation. One important parameter in this step, is how many iterations of smoothing will be performed.

In the rest of the article, the following notation will be used. v_l is the current solution for one component in the GVF vector field $\mathbf{V}$ at resolution level l. r_l is the residual of the current solution v_l at resolution level l. The vector $\mathbf{x} = [x, y, z]$, is the voxel position. The squared magnitude of the initial vector field $\mathbf{V}_0$ is constant and simplified to $S_l(\mathbf{x}) = |\mathbf{V}_0(\mathbf{x})|_l^2$. Assuming isotropic spacing h in the computational grid, the update equation for the Gauss-Seidel method is as follows [9]:

$$\begin{aligned} L(x,y,z) &= v_l(x+1,y,z) + v_l(x-1,y,z) + v_l(x,y+1,z) \\ &\quad + v_l(x,y-1,z) + v_l(x,y,z+1) + v_l(x,y,z-1) \\ v_l(x,y,z) &= \frac{2\mu L(x,y,z) - 2h_l^2 r_l(x,y,z)}{12\mu + h_l^2 S_l(\mathbf{x})} \end{aligned} \tag{4}$$

The update equation is executed on the entire dataset at each level and is done with two kernels as shown in Algorithm 2. To accomplish the checkerboard (red-black) pattern as shown in Figure 3, the Manhatten distance from origo $(x + y + z)$ is first calculated. If the Manhatten distance is even the voxel is red, and if it is odd the voxel is black. The first kernel, GaussSeidelRed, calculates the red voxels using Equation 4. Thus this kernel only works on half the voxels in the dataset. The second kernel, GaussSeidelBlack, copies the red voxels from the previous kernel and computes the black voxels.

A double buffering mechanism is used here with the datasets v_{read} and v_{write}. This is necessary because the data is stored in textures which can only be read or written to in a kernel. More details about the use of textures can be found in the optimization section.

At the boundary of the image, the neighboring voxels needed to calculate L in Equation 4 does not exist. It is desirable to have a zero gradient at the boundary, because a gradient larger than zero at the boundary would diffuse into the rest of the image giving an impression of an edge at the boundary. This can have the effect of forcing the active contours towards the image boundary. There are several ways to implement a zero gradient at the boundary. In this implementation, any voxel that is on the boundary of the image will change its value to the same as the voxel two steps inside the image as shown in Figure 4. For example a voxel with coordinate $x = 0$ uses the value of the voxel with coordinate $x = 2$. Also, a voxel with $x = N - 1$ uses the value of the voxel with coordinate $x = N - 3$. The same applies for the y and z coordinates. The reason for doing it this way is that the calculations are simple.

2.1.2 Restriction

The restriction step downsamples the residual r at level l to a coarser grid (level $l + 1$).

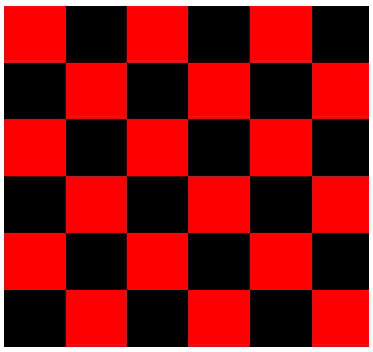

Fig. 3: Checkerboard pattern used in the red-black Gauss-Seidel method to process half the voxels in parallel on the GPU.

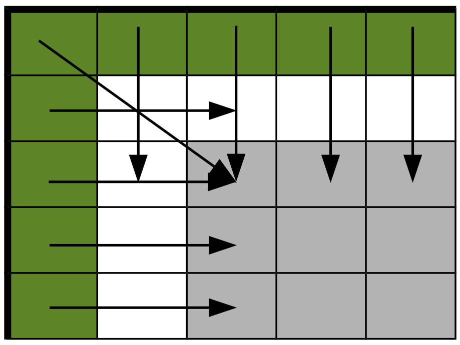

Fig. 4: Illustration of boundary conditions in the top left corner of an image. Boundary pixels (green/dark) get the same value as the pixels two steps inside the image (arrows). This will create zero gradients at the white pixels because a central difference scheme is used for the Laplace operator.

Algorithm 2 Parallel red-black Gauss-Seidel

```
function GaussSeidel($r$, $v$, $i$, $S$, $h$)
    for $i$ times do
        GaussSeidelRed($r$, $v$, $v_t$, $S$, $h$)
        GaussSeidelBlack($r$, $v_t$, $v$, $S$, $h$)
    end for
    return $v$
end function

function GaussSeidelRed($r$, $v_{\text{read}}$, $v_{\text{write}}$, $S$, $h$)
    for each voxel $(x, y, z)$ in parallel do
        if $x + y + z$ is even then
            Use Equation 4 to calculate $v$ for voxel $x, y, z$
        end if
    end for
end function

function GaussSeidelBlack($r$, $v_{\text{read}}$, $v_{\text{write}}$, $S$, $h$)
    for each voxel $(x, y, z)$ in parallel do
        if $x + y + z$ is even then
            Copy the red voxel from $v_{\text{read}}$ to $v_{\text{write}}$
        else
            Use Equation 4 to calculate $v$ for voxel $x, y, z$
        end if
    end for
end function
```

The residual is calculated using the current solution v_l and residual r_l as [9]:

$$r_l(\mathbf{x}) = r_l(\mathbf{x}) - \left(\frac{\mu L(\mathbf{x}) - 6v_l(\mathbf{x})}{h_l^2} - v_l(\mathbf{x}) S_l(\mathbf{x}) \right) \tag{5}$$

The restriction operator used in this implementation takes the average of each 2x2x2 voxel cell and creates a grid which is half the size in each dimension as shown in Equation 6. The same operator is used when creating the different levels of the squared magnitude of the initial vector field S_l.

$$\begin{aligned} r_{l+1}(x,y,z) = &\frac{1}{8}(r_l(2x,2y,2z) + r_l(2x+1,2y,2z) \\ &+ r_l(2x,2y+1,2z) + r_l(2x,2y,2z+1) \\ &+ r_l(2x+1,2y+1,2z) + r_l(2x,2y+1,2z+1) \\ &+ r_l(2x+1,2y,2z+1) + r_l(2x+1,2y+1,2z+1)) \end{aligned} \tag{6}$$

If the grid size is 256x256x256 for level l, the next level will have size 128x128x128. Usually, the size of the finest grid (level $l = 0$), the actual image size, is not equal in every dimension or a power of two for that matter. By taking the largest dimension and rounding up towards the closest number that is a power of 2 (see Equation 7), the size of the next level can be determined.

$$A = 2^{\lceil log2(max(M,N,O)) \rceil - 1} \tag{7}$$

Then the size of level 1 would be AxAxA. Thus for an input vector field of size 460x390x120 (level 0), level 1 would have a size of 256x256x256, and level 2 would have a size of 128x128x128. This method leads to some waste of space and processing, but gives a much simpler implementation. The spacing of each level is calculated as $h_{l+1} = 2h_l$. The multigrid method will process grids from level 0 to the coarsest level with the smallest possible size, 2x2x2.

2.1.3 Prolongation

Prolongation is the opposite of restriction. Prolongation resamples and increases the size of the grid. It is used when correcting the current solution with a solution of a coarser grid such that $\textsc{Correct}(v_l, v_{l+1}) \leftarrow v_l + \textsc{Prolongate}(v_{l+1})$. Bi- or trilinear interpolation may be used as a prolongation operator, but according to Han et al. [9] a simple nearest voxel method (Equation 8) give a better convergence rate.

$$v_l(x,y,z) = v_{l+1}\left(\left\lfloor \frac{x}{2} \right\rfloor, \left\lfloor \frac{y}{2} \right\rfloor, \left\lfloor \frac{z}{2} \right\rfloor \right) \tag{8}$$

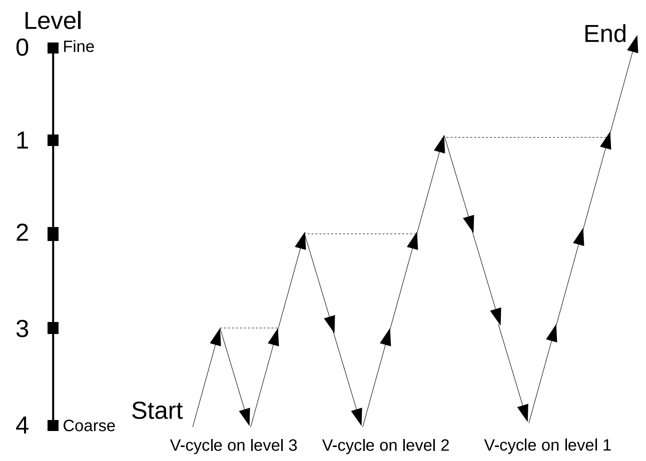

Fig. 5: The full multigrid algorithm with 4 levels.

2.1.4 The V-cycle

Putting all of this together we end up with Algorithm 3. This algorithm needs 6 separate GPU kernels: GaussSeidelRed, GaussSeidelBlack, Residual, Restrict, Correct and InitializeToZero which simply initializes a solution to zero.

The two constants b and c determines how many times pre- and post-smoothing will be performed.

Algorithm 3 The V-cycle

```
function VCYCLE($r_l$, $v_l$, $l$, $S_l$, $h_l$, $b$, $c$)
    $v_l \leftarrow$ GAUSSSEIDEL($r_l, v_l, b, S_l, h_l$)
    if $l$ is NOT the coarsest grid level then
        % Calculate residual of current solution $v_l$
        $r_l \leftarrow$ RESIDUAL($r_l, v_l$)
        $r_{l+1} \leftarrow$ RESTRICT($r_l$)
        % Initialize coarse solution to 0
        $v_{l+1} \leftarrow$ INITIALIZETOZERO
        $v_{l+1} \leftarrow$ VCYCLE($r_{l+1}, v_{l+1}, l+1, S_{l+1}, h_{l+1}, b, c$)
        % Correction of the $v_l$ using the coarse solution
        $v_l \leftarrow$ CORRECT($v_l, v_{l+1}$)
    end if
    $v_l \leftarrow$ GAUSSSEIDEL($r_l, v_l, c, S_l, h_l$)
    return $v_l$
end function
```

2.1.5 The full multigrid algorithm

One MG scheme is to repeat the V-cycle until convergence. However, faster convergence can be achieved with the full MG algorithm (FMG) [9]. The FMG algorithm is based on the MG V-cycle. However, instead of performing a set of similar V-cycles, the FMG algorithm starts with the coarsest grid and uses the solution for this grid to get a good initialization of the next finer grid (see Figure 5). This is done recursively using the function RecursiveFullMultigrid for all computational grids as shown in Algorithm 4. The FMG algorithm can also be repeated until convergence, the constant a determines how many times the FMG algorithm will be repeated. The function FullMultigrid is the entry point of the entire method and takes in the parameters a, b and c and the initial vector field $\mathbf{V}_0$. The FMG algorithm needs one more additional GPU kernel and that is the Prolongation kernel which implements Equation 8. The other kernels, Restrict, Residual and InitializeToZero, are the same as in the V-cycle algorithm.

Algorithm 4 The full multigrid algorithm

```
function RECURSIVEFULLMULTIGRID($r_l$, $l$, $b$, $c$)
    if $l$ is the coarsest grid then
        $v_l \leftarrow$ INITIALIZETOZERO
    else
        $r_{l+1} \leftarrow$ RESTRICT($r_l$)
        $v_{l+1} \leftarrow$ RECURSIVEFULLMULTIGRID($r_{l+1}, l+1, b, c$)
        $v_l \leftarrow$ PROLONGATE($v_{l+1}$)
    end if
    $v_l \leftarrow$ VCYCLE($r_l, v_l, l, S_l, h_l, b, c$)
    return $v_l$
end function

function FULLMULTIGRID($\mathbf{V}_0$, $a$, $b$, $c$)
    $\mathbf{V} \leftarrow$ INITIALIZETOZERO
    for $a$ times do
        for each component $C \in [x, y, z]$ do
            % Calculate initial residual
            $r \leftarrow$ RESIDUAL($-\mathbf{V}_{0,C}|\mathbf{V}_{0,C}|^2, \mathbf{V}_C$)
            $\mathbf{V}_C \leftarrow$ RECURSIVEFULLMULTIGRID($r, 0, b, c$)
        end for
    end for
    return $\mathbf{V}$
end function
```

2.2 GPU optimization

Accessing the off-chip global memory on a GPU is a very time-consuming operation [1]. Since all of the kernels in this implementation require many memory access operations and few arithmetic operations, the performance of these kernels are memory-bound. Thus, optimization of these kernels should focus on optimizing the memory access. In this section, GPU memory optimization techniques such as using the texture memory system and a 16-bit storage format are described.

2.2.1 Texture memory

The GPU has a specialized memory system for images, called the texture system. It has this system because the GPU is primarily made and used for fast rendering which involves mapping images, often called textures, onto 3D objects. The texture system specializes in fetching and caching data from 2D and 3D textures [13,1]. The fetch unit of the texture system is also able to perform interpolation and data type conversion in hardware. When working with images and volumes, using the texture system to store these structures can greatly improve performance as shown in our previous work on GVF [16]. All of the vector fields, residuals and squared magnitudes at different levels are stored in textures.

2.2.2 16-bit storage format

Memory access can also be improved by reducing the number of bytes transferred from global memory to the chip. Floating point numbers are usually represented using 32 bits and the IEEE 754 standard. However, if the floating point numbers are normalized between 0.0 and 1.0 or -1.0 and 1.0 a different format can be used. Most GPU's texture system supports normalized 8- and 16-bit integers. With this format, the data is stored as 8- or 16-bit integers in the textures. However, when the data is requested, the texture fetch unit converts the integer to a 32-bit floating point number with a normalized range. This reduces accuracy, and may not be sufficient for all applications. In our previous work on GPU-based GVF using Euler's method, the results showed that 8-bit was too inaccurate for any practical use [16]. Also, our previous work on applications such as segmentation and centerline extraction of airways and blood vessels using GVF has shown that 16-bit gave just as good results as 32-bit and increased the speed considerably on large images [17,18]. The 16-bit storage format also halves the global memory usage, thus allowing much larger volumes to reside completely in the GPU memory.

2.2.3 Work-group size

Threads are executed on the GPU in groups. AMD calls these units of execution *wavefronts* while NVIDIA calls them *warps* [1,13]. The units are executed atomically and have at the time of writing the size of 64 (AMD) or 32 (NVIDIA) threads. The threads are also grouped in software. In OpenCL these groups are called work-groups, and in CUDA they are called thread-blocks. If the work-group sizes are not a multiple of the wavefront/warp size, some of the GPUs thread processors will be idle for each work-group that is executed. Also, there is a maximum number of threads that can reside in a work-group. On AMD GPUs, this limit is currently 256 and on NVIDIA up to 1024.

When a kernel is scheduled on the GPU using OpenCL, the kernel is executed on a global grid. The grid size has to be dividable by the work-group size. Thus if an image of size 512x512x256 is to be processed with one kernel per voxel, a 3D global execution grid is used with the same size of the image. A possible work-group size is then 4x4x4 because 512 and 256 is dividable by 4, and 4x4x4 = 64 threads which is a multiple of the wavefront/warp sizes and is below the maximum limit.

In this implementation a work-group size of 4x4x4 was used. However, the optimal work-group size can vary from different GPUs. Volumes that have a dimension size that is not dividable by 4 are cropped.

3 Results

The overall goal of the proposed GVF implementation is to achieve a low error as fast as possible. Thus the GVF error and execution time were measured at different number of iterations. This was done on three different datasets using a modern AMD Radeon HD7970 GPU with 3GB memory. The setup was running Ubuntu 12.04, AMD Catalyst 12.11 graphic drivers and AMD APP SDK 2.9. The parameters used in all experiments are $\mu = 0.1$, $a = 1$, $b = 2$ and $c = 1$. Recall that the constant a is the number of times the FMG algorithm is repeated, and b and c are the number of pre- and post-smoothing iterations. These constants were determined through experimentation. The gradient of the input image smoothed with a Gaussian filter with standard deviation $\sigma = 0.5$ was used as the input vector field $\mathbf{V}_0$ in all experiments.

The graphs in Figure 6 show the average error ϵ versus time for both the Euler [16] and multigrid method on the GPU. These measurements were done on three different datasets with varying sizes using both 16- and 32-bit storage. One large volume of size 512x512x512, one medium volume of size 512x512x256 and one small volume of size 256x256x256. All of the volumes are clinical computed tomography (CT) volumes. The average error ϵ is calculated using Equation 2 over all N voxels:

$$\epsilon = \frac{1}{N}\sum_{\mathbf{x}} \left|\mu\nabla^2\mathbf{V}(\mathbf{x}) - (\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x}))|\mathbf{V}_0(\mathbf{x})|^2\right| \quad (9)$$

From these graphs, it is evident that the MG GPU method converges faster than the Euler GPU method for all three datasets. However, it was not possible to process the largest volume (512x512x512) with either method using 32-bit storage as there was not enough memory on the GPU to do this. Thus only results for 16-bit storage are included in the graph for this volume.

Figure 7 shows the increased capture range with the MG method versus the Euler method [16] when run on a CT thorax image using the same amount of execution time. The figure shows images of the magnitude of the GVF vector field $|\mathbf{V}|$ using the same intensity transformation for visual comparison.

Table 1 shows the average runtime on different datasets for the two GPU implementations and a serial C++ CPU implementation of Euler's method. The datasets were processed 10 times with Euler's method first using a fixed number of iterations and then the multigrid GPU implementation was executed for as many iterations as needed to reach the same error ϵ or lower as the Euler

http://github.com/smistad/Gradient-Vector-Flow/

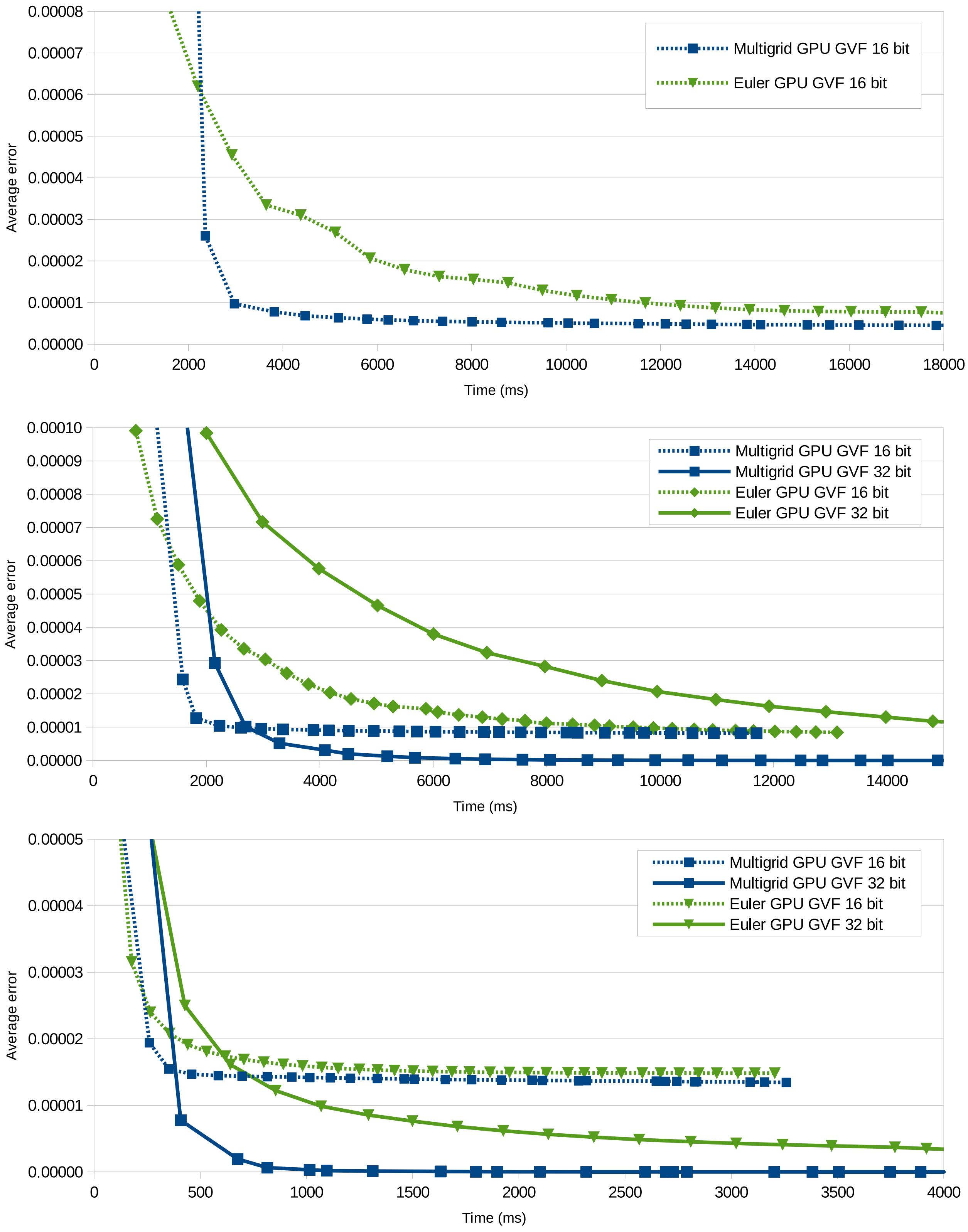

Fig. 6: Average error ϵ over time in ms for both the Euler and multigrid GPU implementations with 32-and 16-bit floating point storage formats and datasets of different sizes. **Top:** 512x512x512. **Middle:** 512x512x256. **Bottom:** 256x256x256.

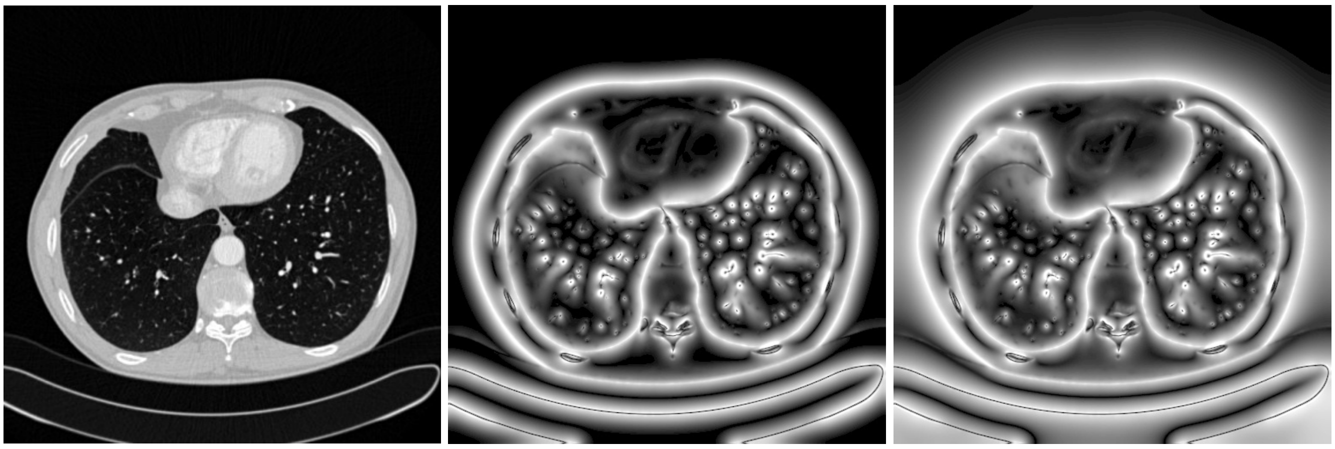

Fig. 7: Magnitude of the GVF vector field after the same amount of execution time displayed using the same intensity transformation. **Left:** Input image (Thorax CT). **Middle:** Euler GPU GVF [16] (512 iterations). **Right:** Multigrid GPU GVF (15 iterations). Note the larger capture range with the multigrid method.

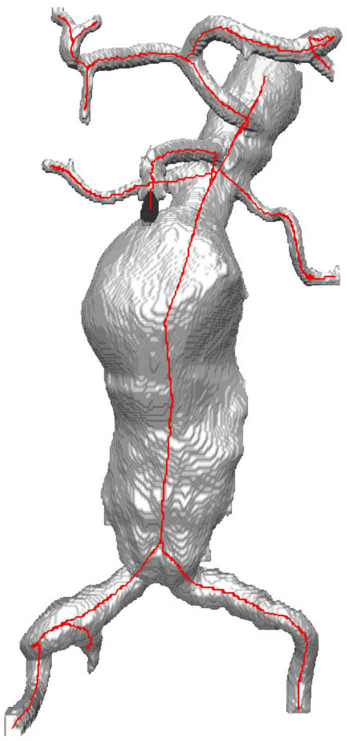

Fig. 8: Segmentation and centerline extraction of an abdominal aortic aneurysm using the proposed multigrid GVF implementation [18].

method. The results show that the multigrid GPU implementation is several times (1.9-5.1) faster than the Euler GPU implementation.

4 Discussion

Defining N as the size of the largest dimension of an image, the Euler method need at least N iterations to diffuse gradients to all voxels of the image. Han et al. [9] defined this as a rule of thumb of how many iterations should be used with this method. This is due to the discrete Laplace operator used which only uses neighbor voxels. Thus, the gradients can only diffuse one voxel at a time. Multigrid methods, on the other hand, can diffuse gradients across the image in a single iteration. This is why multigrid methods for GVF achieve a greater capture range and thus lower error faster. In the experiment depicted in Figure 7, both methods were run for the same amount of time and the result is that the multigrid method ends up with a higher capture range. Although multigrid methods need fewer iterations, the multigrid iterations are much more time consuming as they do a lot more work in each iteration.

In our previous work [18], the proposed MG GVF implementation was used in the segmentation and centerline extraction of abdominal aortic aneurysms (AAAs) (see Figure 8). In this work, GVF is used to diffuse the image gradients from the edge of the blood vessels to the center. Because AAAs often involve very large blood vessels, the gradients have to diffuse a long way and thus benefit a lot from the MG GPU implementation. Using the proposed implementation, 6 iterations and 1-2 seconds of processing were sufficient. While over 10,000 iterations and several minutes of processing were needed to achieve the same result with the Euler GPU implementation using the same GPU.

From the graphs in Figure 6 it is clear that the amount of speedup depends on what the target average error is and the size of the dataset. The speedup may also vary a lot for different types of GPUs. These graphs also show

Dataset size	**Euler iterations**	**Euler CPU** 32-bit	**Euler GPU** 16-bit / 32-bit	**Multigrid GPU** 16-bit / 32-bit	**Multigrid iterations needed**	**Speedup** 16-bit / 32-bit
512x512x512	512	3085 secs	7.80 / N/A secs	2.17 / N/A secs	4	3.6 / N/A
512x512x256	512	1531 secs	3.88 / 10.16 secs	1.4 / 1.99 secs	4	2.8 / 5.1
256x256x256	256	188 secs	0.46 / 1.10 secs	0.24 / 0.33 secs	3	1.9 / 3.3
256x256x128	256	97 secs	0.23 / 0.54 secs	0.12 / 0.17 secs	2	1.9 / 3.2

Table 1: Average runtime for three different GVF implementations: One serial CPU and one GPU implementation of the Euler method, and the proposed multigrid GPU implementation. The Euler method is run with a specific number of iterations for each dataset, and the multigrid method is run for as many iterations needed to reach the same error ϵ or lower.

that it is possible to get a lower average error with 32-bit than with 16-bit storage format.

Note that the multigrid method processes one component of the vector field at a time. This is less efficient than processing all components at the same time as with the Euler method in Algorithm 1. The reason for processing one component at a time in this multigrid implementation is to reduce memory usage. Thus, if a GPU had more memory, all components could be processed in parallel and the method would probably be even faster. As GPUs get more and more memory every year, this will most likely be possible in the near future.

5 Conclusions

In this paper, a GPU implementation of a multigrid solver for gradient vector flow was presented. The results showed that this multigrid implementation was able to achieve a higher capture range with a lower average error faster than a highly optimized GPU implementation of the traditional Euler's method for calculating the gradient vector flow.

References

1. Advanced Micro Devices. AMD Accelerated Parallel Processing OpenCL Programming Guide. Technical Report November, 2013. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf Last accessed 8. August 2013.
2. Rigo Alvarado, Juan J. Tapia, and Julio C. Rolón. Medical image segmentation with deformable models on graphics processing units. *The Journal of Supercomputing*, 68(1):339–364, December 2013.
3. Christian Bauer and Horst Bischof. A novel approach for detection of tubular objects and its application to medical image analysis. In *Proceedings of the 30th DAGM Symposium on Pattern Recognition*, pages 163–172. Springer, 2008.
4. Christian Bauer and Horst Bischof. Extracting curve skeletons from gray value images for virtual endoscopy. In *Proceedings of the 4th International Workshop on Medical Imaging and Augmented Reality*, pages 393–402. Springer, 2008.
5. Jeff Bolz, I Farmer, E Grinspun, and P Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2003*, 22(3):917–924, 2003.
6. O.C. Eidheim, J. Skjermo, and L. Aurdal. Real-time analysis of ultrasound images using GPU. *International Congress Series*, 1281:284–289, May 2005.
7. Harald Grossauer and Peter Thoman. GPU-based multigrid: Real-time performance in high resolution nonlinear image processing. *Computer Vision Systems*, 5008:141–150, 2008.
8. Yujun Guo and Cheng-chang Lu. Multi-modality Image Registration Using Mutual Information Based on Gradient Vector Flow. In *18th International Conference on Pattern Recognition (ICPR'06)*, pages 697–700. Ieee, 2006.
9. X Han, C Xu, and J.L. Prince. Fast numerical scheme for gradient vector flow computation using a multigrid method. *Image Processing, IET*, 1(1):48–55, 2007.
10. M.S. Hassouna and A.A. Farag. On the extraction of curve skeletons using gradient vector flow. In *IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE, 2007.
11. Zhiyu He and Falko Kuester. GPU-Based Active Contour Segmentation Using Gradient Vector Flow. In *Advances in Visual Computing*, pages 191–201, 2006.
12. Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, January 1988.
13. NVIDIA. OpenCL Best Practices Guide. Technical report, 2010. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf Last accessed 8. August 2013.
14. Nilanjan Ray and Scott T Acton. Motion gradient vector flow: an external force for tracking rolling leukocytes with shape and size constrained active contours. *IEEE transactions on medical imaging*, 23(12):1466–78, December 2004.
15. Erik Smistad, Anne C. Elster, and Frank Lindseth. GPU-Based Airway Segmentation and Centerline Extraction for Image Guided Bronchoscopy. In *Norsk informatikkonferanse*, pages 129–140. Akademika forlag, 2012.
16. Erik Smistad, Anne C. Elster, and Frank Lindseth. Real-time gradient vector flow on GPUs using OpenCL. *Journal of Real-Time Image Processing*, pages 1–8, 2012.
17. Erik Smistad, Anne C. Elster, and Frank Lindseth. GPU accelerated segmentation and centerline extraction of tubular structures from medical images. *International Journal of Computer Assisted Radiology and Surgery*, 9(4):561–575, 2014.
18. Erik Smistad and Frank Lindseth. A New Tube Detection Filter for Abdominal Aortic Aneurysms. In *Proceedings of MICCAI 2014 Workshop on Abdominal Imaging: Computational and Clinical Applications*, 2014.
19. Chenyang Xu and J.L. Prince. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing*, 7(3):359–369, 1998.
20. Zuoyong Zheng and Ruixia Zhang. A Fast GVF Snake Algorithm on the GPU. *Research Journal of Applied Sciences, Engineering and Technology*, 4(24):5565–5571, 2012.