J Real-Time Image Proc
DOI 10.1007/s11554-012-0257-6

ORIGINAL RESEARCH PAPER

Real-time gradient vector flow on GPUs using OpenCL

Erik Smistad · Anne C. Elster · Frank Lindseth

Received: 23 December 2011 / Accepted: 18 May 2012

Abstract The Gradient Vector Flow (GVF) is a feature-preserving spatial diffusion of gradients. It is used extensively in several image segmentation and skeletonization algorithms. Calculating the GVF is slow as many iterations are needed to reach convergence. However, each pixel or voxel can be processed in parallel for each iteration. This makes GVF ideal for execution on Graphic Processing Units (GPUs). In this paper, we present a highly optimized parallel GPU implementation of GVF written in OpenCL. We have investigated memory access optimization for GPUs, such as using texture memory, shared memory and a compressed storage format. Our results show that this algorithm really benefits from using the texture memory and the compressed storage format on the GPU. Shared memory, on the other hand, makes the calculations slower with or without the other optimizations because of an increased kernel complexity and synchronization. With these optimizations our implementation can process 2D images of large sizes (512^2) in real-time and 3D images (256^3) using only a few seconds on modern GPUs.

Keywords Gradient Vector Flow · GPU · OpenCL

E. Smistad (✉) · A. C. Elster · F. Lindseth
Department of Computer and Information Science,
Norwegian University of Science and Technology,
Sem Saelandsvei 7-9, 7491 Trondheim, Norway
e-mail: smistad@idi.ntnu.no

F. Lindseth
SINTEF Medical Technology, Trondheim, Norway

1 Introduction

The Gradient Vector Flow (GVF) is a feature-preserving spatial diffusion of gradients. The GVF field is defined as the vector field $\mathbf{V}$, that minimizes the energy function E:

$$E(\mathbf{V}) = \int \mu|\nabla\mathbf{V}(\mathbf{x})|^2 + |\mathbf{V_0}(\mathbf{x})|^2|\mathbf{V}(\mathbf{x}) - \mathbf{V_0}(\mathbf{x})|^2\mathbf{dx} \quad (1)$$

where $\mathbf{V_0}$ is the initial vector field.

The GVF was introduced by Xu and Prince [11] 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 its energy and are used extensively for boundary detection and segmentation. The traditional snake introduced by Kass et al. [8] has the problem of getting stuck in boundary concavities and low capture range. The GVF snake can deal with these problems.

Figure 1 depicts the GVF when used for active contours. The initial image shown top-right is an image smoothed by convolution with a Gaussian. Next is the initial vector field $\mathbf{V_0}$ displayed using vector magnitude in the top row and the vectors in a zoomed region below. The next column shows the GVF field after 10 iterations of diffusion and the last column 400 iterations.

After its introduction, the GVF has been applied on several other image processing applications. Bauer and Bischof [2] developed a novel approach to use the GVF as a replacement for the scale-space framework in Hessian-based tube detection. Hassouna and Farag [6] and Bauer and Bischof [3] used the GVF to extract skeletons from objects. Ray and Acton [10] used GVF to track leukocytes from intravital video microscopy. Guo and Lu [4] argued that GVF combined with Mutual Information can improve multi-modal image registration.

Published online: 05 June 2012

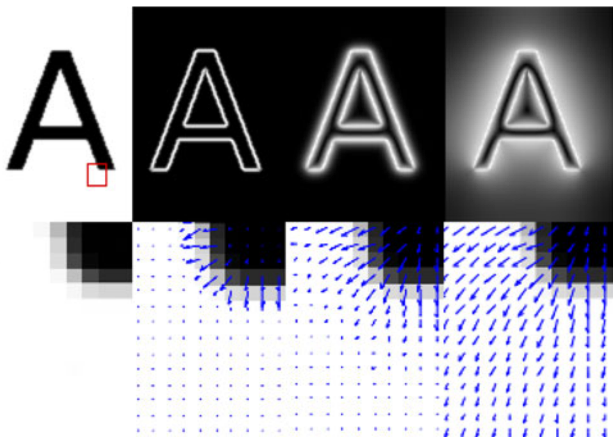

Fig. 1 Example of GVF execution. From *left to right top* 1 smoothed image, 2 magnitude of image gradients $\mathbf{V_0}$, 3 magnitude of GVF after 10 iterations, 4 magnitude of GVF after 400 iterations. *Bottom* 1 zoomed area of smoothed image 2, 3 and 4 image gradients superimposed on zoomed image after 0, 10 and 400 iterations

Xu and Prince [11] 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 is done by treating the vector field $\mathbf{V}$ as a function of time. Calculating the GVF field serially using this numerical approach is slow due to the need for many iterations to reach convergence. However, since each pixel is calculated independently of the other pixels, each pixel can be processed in parallel with the exact same instructions for each iteration. This data parallelism makes the GVF ideal for running on Graphic Processing Units (GPUs). GPUs enable execution of the same instructions on many different data elements in parallel.

He and Kuester [7] presented a GPU implementation of GVF and Active Contours using OpenGL Shading Language (GLSL). They reported that their GPU implementation was up to 4 times faster than a CPU implementation. Their implementation was for 2D images only and used the texture memory system to speed up data retrieval. Performance result for only one NVIDIA GPU was presented. Also, Han et al. [5] proposed another serial numerical scheme for GVF using a multigrid method. Their results showed significant improvement in speed.

In this paper, we present an optimized parallel GVF implementation written in OpenCL. OpenCL is a new cross-platform framework for writing applications that can run on heterogeneous systems. In contrast to the work of He and Kuester [7], we investigate three different memory optimization techniques for GPUs instead of just using the texture memory. We also discuss 3-dimensional GVF and show results for both GPUs and multi-core CPUs from different manufacturers.

In the next section, we show how GVF can be implemented in parallel and note that the algorithm is memory intensive. We also present three memory usage optimizations for GPUs: texture memory, shared memory and a 16-bit floating point data type for storage. Section 3 presents performance results for each optimization in terms of both speed and memory usage. An analysis of the accuracy of the 16-bit floating point data type is also conducted. Section 4 provides a discussion of the presented results and the last section conclusions.

2 GPU implementation

The parallel version of the numerical implementation of GVF by Xu and Prince [11] is given in Algorithm 1 and for 3D in Algorithm 2. The Laplacian $\nabla^2\mathbf{V}(\mathbf{x})$ is calculated using a finite difference method. On the boundaries of the image, some of the neighboring points required to calculate the Laplacian, will not exist. This can be solved by expanding the image with 1 pixel in all directions and have the same vector on the border as the third outermost pixel as depicted in Fig. 2. The gradient at the original border will then be 0. In practice, this is done by swapping the *x*, *y* or z components in the read address to 2 if it is 0 and to M-2 if it is M, where M is the size of that dimension.

Algorithm 1 Parallel 2D Gradient Vector Flow

for all points $\mathbf{x}$ in parallel **do**
  laplacian $\leftarrow -4\mathbf{V}(\mathbf{x})+\mathbf{V}(x+1,y)+\mathbf{V}(x-1,y)+\mathbf{V}(x,y+1)+\mathbf{V}(x,y-1)$
  $\mathbf{V}(\mathbf{x}) \leftarrow \mathbf{V}(\mathbf{x}) + \mu*$ laplacian $-(\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x}))|\mathbf{V}_0(\mathbf{x})|^2$
end for

Algorithm 2 Parallel 3D Gradient Vector Flow

for all points $\mathbf{x}$ in parallel **do**
  laplacian $\leftarrow -6\mathbf{V}(\mathbf{x}) + \mathbf{V}(x+1,y,z) + \mathbf{V}(x-1,y,z) + \mathbf{V}(x,y+1,z)+\mathbf{V}(x,y-1,z)+\mathbf{V}(x,y,z+1)+\mathbf{V}(x,y,z-1)$
  $\mathbf{V}(\mathbf{x}) \leftarrow \mathbf{V}(\mathbf{x}) + \mu*$ laplacian $-(\mathbf{V}(\mathbf{x}) - \mathbf{V}_0(\mathbf{x}))|\mathbf{V}_0(\mathbf{x})|^2$
end for

From these pseudocodes, we can see that calculating the GVF needs 6 global memory accesses for 2D and 8 for 3D and about 20 ALU operations. The GVF computation is memory-bound because global memory access can have a latency of several hundred clock cycles while the ALU operations are only a small fraction of this [1]. Thus, in this project, we have focused on optimizing memory access and storage.

The unoptimized GPU implementation uses regular global memory with a 32-bit floating point storage format.

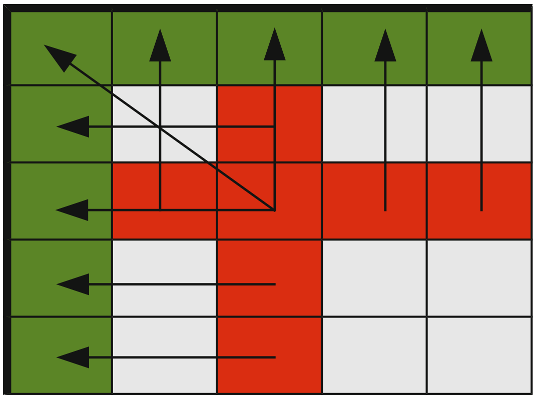

Fig. 2 The *top left corner* of an image. The *arrows* indicate the values the boundary pixels use

In this article, we explore using texture memory as an alternative to global memory as well as shared memory in combination with texture and global memory. We also use a compressed 16-bit floating point storage format with each of these 4 memory combinations as an alternative to the default 32-bit format. Thus in total, we test eight different memory optimization combinations on the GPU.

2.1 Texture memory

The default memory on GPUs is called global memory. This memory is not always cached (for AMD GPUs, global memory caching has to be enabled explicitly). When caching is enabled, it only has linear spatial locality. Most modern GPUs also have a separate texture memory system. Textures are 1D, 2D or 3D structures that can be addressed based on coordinates. GPUs have this texture memory system because GPUs are primarily used for 3D applications where textures are mapped to 3D objects to create a more realistic 3D scene. The textures are stored off-chip, but are cached and have spatial locality in multiple dimensions. When working with images and volumes this cache with 2D/3D spatial locality can increase cache hits.

In the GVF calculations, there are two 2D/3D structures: the GVF field $\mathbf{V}$ and the initial vector field $\mathbf{V_0}$. We optimize our implementation by putting both of these data structures in textures. In OpenCL, textures are called images, and an image bound to a kernel can only be either read or written to. This is a limitation needed to assure cache coherency. Since the GVF vector field $\mathbf{V}$ has to be both read and written, we have used a double buffering mechanism.

By creating two textures for the GVF field $\mathbf{V}$, we use one texture for writing and one for reading, and after each iteration we swap the textures in the arguments to the kernel.

The handling of the boundaries as depicted in Fig. 2 can be handled automatically by the texture system using the addressing flag *ADDRESS_CLAMP_TO_EDGE*. With this flag set, pixels requested outside of the texture will use the pixel value closest to the request pixel.

In OpenCL, writing to a 3D texture is an optional extension called *cl_khr_3d_image_writes*. AMD supports it while NVIDIA does not. To support 3D GVF calculation on NVIDIA GPUs we created a separate kernel for these devices that uses global memory instead of textures for $\mathbf{V}$. Since global memory only have linear spatial cache locality, this is expected to reduce the number of cache hits.

2.2 Shared memory

Shared memory is an on-chip memory that is shared among all work items in a work group. This memory is reported by GPU manufacturers to be more than 10 times faster than global memory which is off-chip [1, 9]. It is generally beneficial to use shared memory when several work items need the same data from global memory as their neighboring work items.

When calculating the Laplacian, $\nabla^2\mathbf{V}(\mathbf{x})$, the data from the 4 (or 6 for 3D) closest neighboring pixels are needed. If N is the total number of pixels, there will be $5N$ global memory accesses to $\mathbf{V}$ in total because each pixel is requested 5 times. By using shared memory the number of global memory accesses can be reduced significantly.

The input image is divided into a set of work groups as shown in Fig. 3. Each work group process one tile of the input image and allocates a block of shared memory with the same size as the work group. Each work item in a work group loads the pixel value from global memory and stores it in shared memory. As the work items on the edges of the work group will not have all their neighbor's data in shared memory, these work items will not do calculations, only load data. These pixels are called the work group's *frame* and are calculated by their neighboring work groups. This causes some overhead in terms of redundant global memory accesses and work items that are idle, but this is very small compared to the overhead of $5N$ global memory accesses to $\mathbf{V}$.

Synchronization is necessary after writing to the shared memory, because all work items in a work group are not executed simultaneously (if a work group is above a certain size). Work items in a work group can synchronize using a barrier in the shared memory.

The shared memory is divided into several banks usually 16 or 32. Memory requests to different banks can be served in parallel while memory requests to the same bank has to be serialized. Requests to the same bank in a clock cycle is called a bank conflict. These bank conflicts can be avoided with a sequential access pattern.

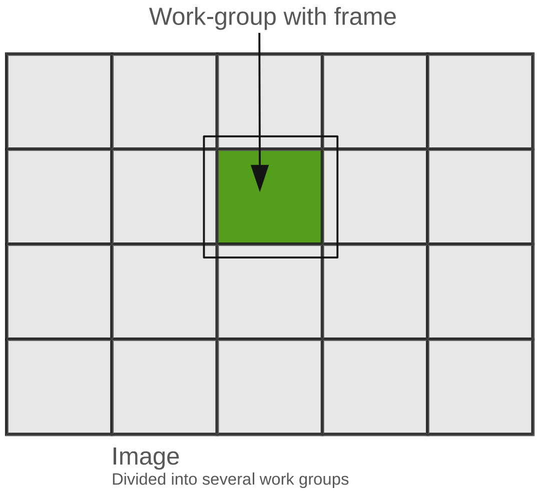

Fig. 3 The input image is divided into several work groups. The *green/dark* area is the part of the work group that is calculated and the box around is the frame where only data is loaded

2.3 16-bit float storage format

Memory access can also be improved by reducing the number of bytes transferred from global memory to the chip. The most common way to store a floating point number on a computer, at present time, is by using 32 bits with the IEEE 754 standard. However, most GPUs also support a texture storage format called normalized 16-bit integer. With this format, the data is stored as 16-bit integers (shorts) in textures, but when it is requested, the texture fetch unit converts the 16-bit integer to a 32-bit floating point number with a normalized range from -1.0 to 1.0. This reduces accuracy, and may not be sufficient for all applications. Due to the reduced accuracy, the 16-bit storage format is made optional in our implementation. This storage format also halves the global memory usage, thus allowing much larger 3D volumes to reside completely in the GPU memory.

2.4 Work-group sizes

Work items are executed on the GPU in groups. AMD calls these units of execution *wavefronts* while NVIDIA calls them *warps*. The units are executed atomically and has at the time of writing the size of 32 or 64 work items. If the work-group sizes are not a multiple of this size, some of the GPUs stream processors will be idle for each work group that is executed. There is also a maximum number of many work items that can exists in one work group. On AMD GPUs, this limit is currently 256 and on NVIDIA up to 1,024. In conjunction with shared memory, we want to maximize the size of the work group minus the frame, given this limit. For 2D, this is maximum when the work group is 16×16 and for 3D, $8 \times 8 \times 4$; e.g., an image of size 512×512 would give 32×32 work groups of size 16×16. Also, in OpenCL, each dimension has to be dividable by the work-group size. Thus, we pad the data so that the size is dividable by the highest possible work group. This avoids idle threads and branch divergence while keeping a large work-group size.

3 Results

3.1 Speed

The speed of our implementation was measured using OpenCL timers. Figure 4 shows the average execution time of one iteration on an image of size 512×512 with different combinations of global, texture and shared memory as well as 32- and 16-bit storage formats. This figure clearly shows that using the texture memory is faster than using regular global memory. Also, it illustrates that utilizing shared memory slows down the computation and that the 16-bit storage format is only beneficial when used together with the texture memory. Figure 5 shows the average total execution time for images of different sizes for both 32- and 16-bit. In this figure, we notice that as the image size increases, the execution time difference also increases. All of these tests were run on an AMD Radeon HD5870 with 1GB of memory.

Tables 1 and 2 includes the average execution time measured both on 2D and 3D and on several different GPUs and multi-core CPUs. For the GPUs only the texture memory with the 16-bit storage format was used. For the CPUs the same version was used, but with 32-bit instead. From these two tables, we observe two things: (1) execution on GPUs is much faster than on CPUs; (2) while NVIDIAs GPUs are comparable to AMDs GPUs on the 2D dataset in terms of speed, NVIDIAs GPUs perform much worse on the 3D dataset.

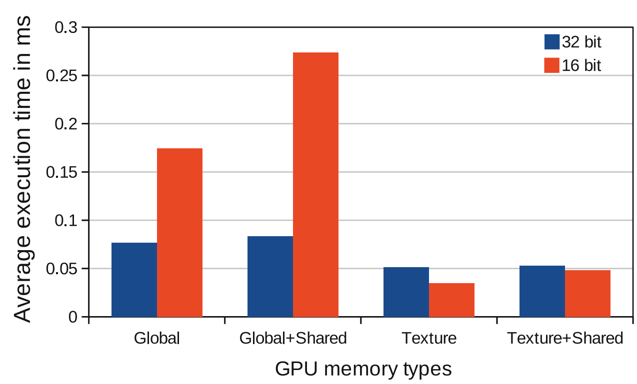

Fig. 4 Average execution time for one iteration of a 512×512 image measured in milliseconds using OpenCL timers with both 32- and 16-bit storage format and different combinations of using regular global memory, texture memory and shared memory

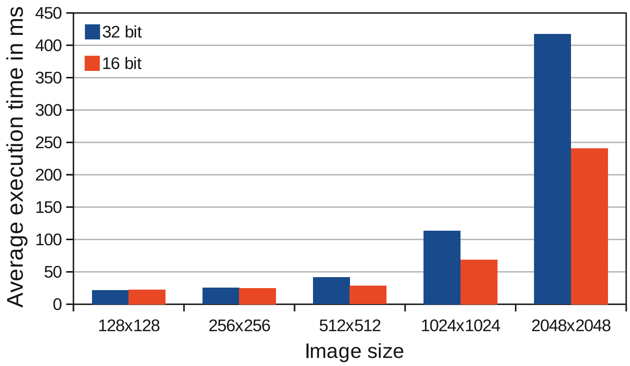

Fig. 5 Average execution time for 512 iterations of images of different sizes using OpenCL timers with both 32- and 16-bit storage format. The execution time difference between 32- and 16-bit storage format increases with the size of the images

3.2 Memory usage

Global synchronization is needed in each iteration when calculating GVF in parallel. Because global synchronization is not possible inside a kernel, a double buffering mechanism is needed. This means that two copies of the vector field **V** is needed in addition to the initial vector field $\mathbf{V_0}$. The GPU implementation needs 2 vector components (x and y) $\times$ 3 vector fields $\times$ 32 bits = 24 bytes per pixel and 36 bytes per voxel for 3D volumes, because of the additional z component. On the other hand, when using a 16-bit float storage format, the memory usage is halved. As an example a volume of size 512^3 would consume 4.5 GB with the 32-bit data type and only 2.25 GB with the 16-bit data type. Figures 6 and 7 graphs the memory usage for this implementation for images and volumes for both 32 and 16-bit. Both figures depict the fact that the difference in memory usage increases as the dataset size increases.

3.3 Relative accuracy

We measured the relative error between a 32 and a 16-bit floating point data type on the final GVF vector field of the

Table 1 Average execution speeds for a 2D image of size 512 $\times$ 512 run for 512 iterations

Processor	One iteration (ms)	All iterations (ms)
AMD 5870	0.035	28
AMD Mobile 5830	0.147	77
NVIDIA Quado FX5800	0.104	66
NVIDIA Tesla c2070	0.077	41
Intel i5 750	1.485	851
Intel i7 720	2.344	1,550

The first 4 processors are GPUs, while the rest are multi-core CPUs

Table 2 Average execution speeds for a 3D volume of size 256^3 run for 256 iterations

Processor	One iteration (ms)	All iterations (ms)
AMD 5870	4.501	1,124
AMD Mobile 5830	20.739	5,129
NVIDIA Quadro FX5800	105.631	2,7172
NVIDIA Tesla c2070	27.989	7,151
Intel i5 750	310.846	92,591
Intel i7 720	378.876	106,747

The first 4 processors are GPUs, while the rest are multi-core CPUs

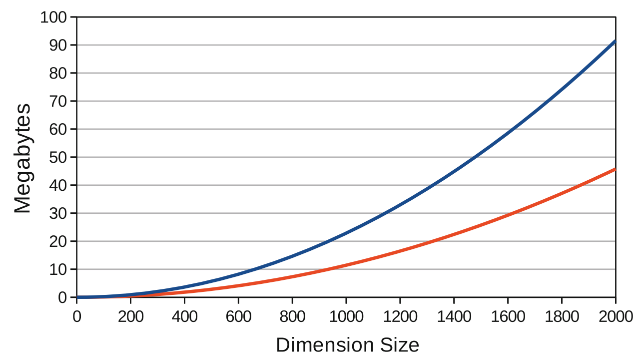

Fig. 6 Memory usage in MBs versus size of image. Dimension size x on the x axis is the size of one of the dimensions so that total number of pixels is x^2

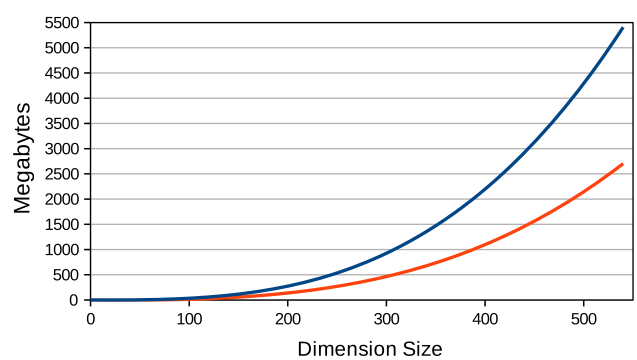

Fig. 7 Memory usage in MBs versus size of volume. Dimension size x on the x axis is the size of one of the dimensions so that total number of voxels is x^3

512 $\times$ 512 image shown in Fig. 8. This was done by calculating the GVF for each data type on the same image. Relative error measures for both the magnitude and angle were calculated as shown in Eqs. 3 and 4. From these equations, the average, variance, maximum and minimum were calculated for all pixels **x** and collected in Table 3.

$$M_{\text{error}} = ||\mathbf{V}_{\mathbf{16}\text{bit}}(\mathbf{x})| - |\mathbf{V}_{\mathbf{32}\text{bit}}(\mathbf{x})|| \tag{3}$$

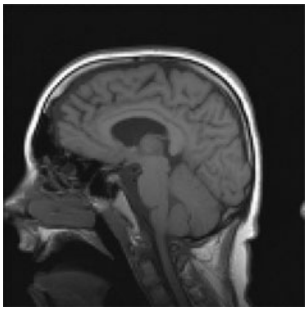

Fig. 8 The 512 × 512 MRI brain scan image the relative error measurements have been run on

Table 3 Relative error of vector magnitude M and angle θ from 32-bit to 16-bit floating point storage format

	M_{error}	θ_{error}
Average	0.00078	0.55
Variance	4.29e−7	0.59
Maximum	0.00377	3.14
Minimum	8.92e−10	0

Calculated using Eqs. 3 and 4 on the image in Fig. 8. Angles are in radians

$$\theta_{\text{error}} = cos^{-1}\left(\frac{\mathbf{V}_{\mathbf{16}\text{bit}}(\mathbf{x}) \cdot \mathbf{V}_{\mathbf{32}\text{bit}}(\mathbf{x})}{|\mathbf{V}_{\mathbf{16}\text{bit}}(\mathbf{x})||\mathbf{V}_{\mathbf{32}\text{bit}}(\mathbf{x})|}\right) \tag{4}$$

4 Discussion

4.1 Speed

Figure 4 shows that introducing shared memory actually makes the calculations slower. The reason for this is threefold: the code is more complex, requires explicit work-group synchronization and more threads/work items are needed. Also, we notice that using the texture memory on the GPU is much faster than using the global memory, which is due to the 2D/3D caching.

This figure further shows that using the 16-bit storage format without textures is slower than using the 32-bit storage format. When the 16-bit format is used in conjunction with textures on GPUs all the data type conversions are done in hardware in the texture fetch units which is much faster than doing the conversion in the code. With CPUs using 32 bits is faster than 16 bits because although the CPU supports texture structures in OpenCL, the CPU does not have dedicated texture fetch units that can do the data type conversion in hardware as GPUs do.

Also, we noticed from Tables 1 and 2 that NVIDIAs GPUs performed much worse on the 3D dataset than AMDs GPUs. The reason for this is that NVIDIA does not support writing to 3D textures in their OpenCL implementation. Thus, global memory had to be used. This memory, as we have explained earlier, is much slower than the texture memory.

Figure 5 illustrates that the difference in execution time between using 32- and 16-bit storage formats increases as the image size increases. Thus the performance gain for 16-bit is biggest for large images and volumes, while for very small images it is almost insignificant.

4.2 Memory usage

From the graph in Fig. 6, we can see that processing 2D images of typical sizes is no problem with modern GPUs that have 1GB memory and more. For 3D volumes a 1GB graphics card would manage to process a dataset, without any additional PCI express data transfer, of about 300^3 and 380^3 voxels for 32- and 16-bit data types, respectively.

4.3 Relative accuracy

Relative accuracy tests were performed to measure the error by using the 16-bit storage format versus 32-bit. As seen in Table 3 these tests showed that there was very little error in magnitude, but on average around 30° angle error. The high angle errors was found to only be present for the very short vectors. In fact, the maximum magnitude of all vectors with angle error above 0.1 was 9.15×10^{-4} on the 512 × 512 MRI brain scan image. The size of the angle error generally increases when the vector length decreases. Thus, this angle error may not be problematic for most applications. For instance, very short vectors will have very little pulling force on a snake.

Still, the capture range of using the 16-bit format is lower than 32-bit as seen in Fig. 9 where the resulting vector field has been normalized. Thus, the 16-bit storage format may not be sufficient for all applications.

5 Conclusions

In this paper, we presented a highly optimized parallel GPU implementation of Gradient Vector Flow written in OpenCL.[1] Our implementation enables real-time execution of GVF for images of sizes up to 512^2 on modern GPUs. Since it is written in OpenCL, it can also run efficiently on multi-core CPUs. We investigated three different memory optimizations for GPUs. Our results show that using the texture memory with the 16-bit compressed floating point storage format and without shared memory is fastest on

[1] The source code of this implementation is available online at http://www.github.com/smistad/OpenCL-GVF/.

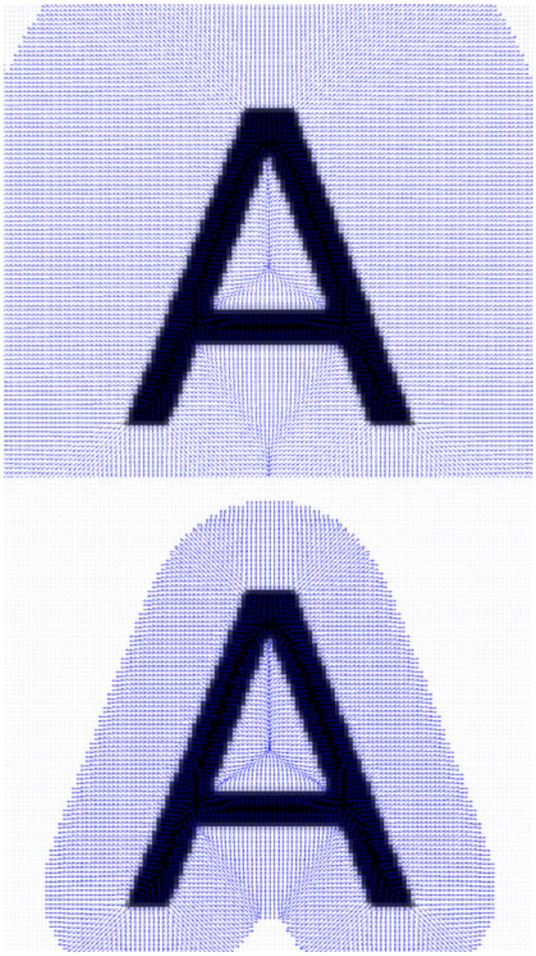

Fig. 9 Normalized GVF vector field, run with the same number of iterations. The *top* is with 32-bit storage format and the *bottom* is 16 bit. These two images clearly show the reduced capture range when using 16 bit

GPUs and can double the performance compared to an unoptimized GPU implementation. Relative accuracy measurements reveal that there is very little error in magnitude, but a high angle error between the 32- and 16-bit storage formats. However, the high angle errors are only present on very small vectors, and thus may not be a problem for most applications. The 16-bit storage format has also the advantage of allowing much larger volumes to reside completely in the limited memory on GPUs.

Acknowledgments Great thanks goes to the people of the High Performance Computing Lab at NTNU for all their assistance and to Mai Britt Engeness Mørk for her comments on the manuscript. The authors would also like to convey thanks to NTNU, NVIDIA and AMD. Without their hardware contributions to the HPC Lab, this project would not have been possible.

References

1. AMD: AMD APP OpenCL programming guide. Technical report, AMD (2011) (http://developer.amd.com/sdks/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf). Last accessed December 2011
2. Bauer, C., Bischof H.: A novel approach for detection of tubular objects and its application to medical image analysis. Pattern Recognit. pp. 163–172 (2008)
3. Bauer, C., Bischof, H.: Extracting curve skeletons from gray value images for virtual endoscopy. Medical Imaging and Augmented Reality, pp. 393–402 (2008)
4. Guo, Y., Lu, C.-C.: Multi-modality image registration using mutual information based on gradient vector flow. In: 18th International Conference on Pattern Recognition (ICPR'06), pp. 697–700 (2006)
5. Han, X., Xu C., Prince, J.L.: Fast numerical scheme for gradient vector flow computation using a multigrid method. IET Image Process. **1**(1), 48–55 (2007)
6. Hassouna, M.S., Farag A.A.: On the extraction of curve skeletons using gradient vector flow. In: 2007 IEEE 11th International Conference on Computer Vision, pp. 1–8 (2007)
7. He, Z., Kuester, F.: GPU-based active contour segmentation using gradient vector flow. In: Advances in Visual Computing, pp. 191–201 (2006)
8. Kass, M., Witkin, A., Terzopoulos, D.: Snakes: Active contour models. Int. J. Comput. Vis. **1**(4), 321–331 (1988)
9. NVIDIA: OpenCL best practices guide. technical report, 2009. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf. Last accessed December 2011
10. Ray, N., Acton, S.T.: Motion gradient vector flow: An external force for tracking rolling leukocytes with shape and size constrained active contours. IEEE Trans. Med. Imaging **23**(12): 1466–1478 (2004)
11. Chenyang, X.u., Prince, J.L.: Snakes, shapes, and gradient vector flow. IEEE Trans. Image Process. **7**(3), 359–369 (1998)

Author Biographies

Erik Smistad is currently a PhD Candidate at the Computer and Information Science department at the Norwegian University of Science and Technology. His thesis is entitled Medical Image Segmentation for Improved Surgical Navigation. In his thesis, one of his goals is to investigate the use of GPU computing to improve the speed of medical image segmentation algorithms.

Dr. Anne C. Elster has been an Associate Professor of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway since 2001 where her research focuses on heterogeneous and parallel computing, including GPU computing. She also holds a Visiting Scientist appointment at University of Texas at Austin's ECE Department. Dr. Elster holds a BSc in CSE from University of Massachusetts at Amherst, and MS and PhD (1994) degrees in EE from Cornell University. She worked for Schlumberger in Austin before returning to acadmia via the University of Texas at Austin in 1997. She served on the MPI standards committees (MPI and MPI-2), and became a Senior Member of the IEEE in 2000. Dr. Elster helped in finding NTNU's Computational Science and Engineering Program where she served as Co-Director until January 2007, and is currently one of four Working Group leaders of EU COST Action IC0805: Open European Network for High Performance Computing on Complex Environments.

Dr. Frank Lindseth is a Senior Research Scientist at SINTEF Medical Technology working in the field of surgical navigation within the national center for ultrasound and image guided therapy since 1996, in close collaboration with surgeons at St. Olavs university hospital in Trondheim, Norway. He is also an Adjunct Associate Professor at the department of Computer and Information Science, the Norwegian University of Science and Technology (NTNU) with special focus on medical image analysis and visualization. Dr. Lindseth received his PhD in ultrasound-guided minimally invasive surgery in 2002 and worked with a project called the surgical navigation system of the future during his PostDoc, both at SINTEF/NTNU.