Accelerating Total Variation Regularization for Matrix-Valued Images on GPUs

Maryam Moazeni
Computer Science Department
University of California, Los Angeles
mmoazeni@cs.ucla.edu

Alex Bui
Department of Radiological Sciences
University of California, Los Angeles
buia@mii.ucla.edu

Majid Sarrafzadeh
Computer Science Department
University of California, Los Angeles
majid@cs.ucla.edu

ABSTRACT

The advent of new matrix-valued magnetic resonance imaging modalities such as Diffusion Tensor Imaging (DTI) requires extensive computational acceleration. Computational acceleration on graphics processing units (GPUs) can make the regularization (denoising) of DTI images attractive in clinical settings, hence improving the quality of DTI images in a broad range of applications. Construction of DTI images consists of direction-specific Magnetic Resonance (MR) measurements. Compared with conventional MR, direction-sensitive acquisition has a lower signal-to-noise ratio (SNR). Therefore, high noise levels often limit DTI imaging. Advanced post-processing of imaging data can improve the quality of estimated tensors. However, the post-processing problem is only made more computationally difficult when considering matrix-valued imaging data.

This paper describes the acceleration of a Total Variation regularization method for matrix-valued images, in particular, for DTI images on NVIDIA Quadro FX 5600. The TV regularization of a 3-D image with 128^3voxels ultimately achieves 266X speedup and requires 1 minute and 30 seconds on the Quadro, while this algorithm on a dual-core CPU completes in more than 3 hours. In this application study we are aimed at analyzing the effective of excessive synchronization, which provides an insight into generally adapting Variational methods to the GPU architecture for other image processing algorithms designed for matrix-valued images.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; I.3.1 [**Computing Methodologies**]: Computer Graphics—*Hardware Architecture*

CF'09, May 18–20, 2009, Ischia, Italy.
Copyright 2009 ACM 978-1-60558-413-3/09/05 ...$5.00.

General Terms

Algorithms, Performance

Keywords

GPGPU, GPU computing, CUDA, Variational methods, TV Regularization

1. INTRODUCTION

Traditional single-core microprocessors have driven rapid performance increase in two decades. However, constraints on power consumption slowed this progress, which have forced CPU vendors to find other ways to meet high performance computing needs in science and engineering. One solution is that of many-core architectures, which integrates tens or hundreds of processors onto a single chip. Many-core processors can offer higher performance or power efficiency compared to current CPU or multi-core processors [1].

One example of commodity many-core processors are the current programmable graphics processing units (GPUs) such as AMDR580 or NVIDIA G80 GPU's with CUDA [2] created as their programming model. Current GPU's have hundreds of processor cores and high memory bandwidth. This processing power of GPU's has been successfully exploited in the GPGPU domain, especially in scientific, imaging and database applications. Fully programmable cores in modern GPUs support important features for general-purpose computing such as IEEE floating point arithmetic, multiprocessor shared memory, and arbitrary memory addressing [2, 3]. Furthermore, increasing the flexibility and programmability of GPU's in recent years has improved their suitability for high-performance computing. For example, in G80, general-purpose applications are developed using ANSI C with extensions provided by CUDA. Previously, GPU programming required the use of graphics APIs [4, 5] or a high-level language on top of graphics API [6, 7, 8]. Developers prefer GPUs over other alternative parallel processors due to several advantages including their low cost and wide availability.

A large number of medical imaging algorithms, including all the algorithms in the medical imaging pipeline (i.e. denoising, registration and segmentation) can benefit significantly from accelerators such as GPUs. During the last decade, a new magnetic resonance modality called diffusion tensor imaging (DTI) has been extensively studied [9, 10, 11, 12, 13, 14]. The DTI images are matrix valued. In each voxel of the imaging domain, a diffusion tensor (i.e. diffusion matrix) D is constructed based on a series of K direction-specific MR measurements. All measurements contain noise, which degrades the accuracy of the estimated tensor. Compared with conventional MR, direction-sensitive acquisition has a lower signal-to-noise ratio (SNR). There are several ways to increase the accuracy of estimated tensors. The most intuitive way is to make an average of a series of repeated measurements. Alternatively, the number of gradient directions can be increased. An obvious disadvantage of both of these approaches is the increased scanner time. Best way to improve the quality of the tensor is by post-processing the data.

Hence, from the developments in DTI, there is a need for robust regularization (denoising) methods for matrix-valued images that is computationally fast. One of the state-of-the art methods for regularization of tensor-valued images is proposed in [15] as a Variational method [16, 17, 18] and a natural extension of the color Total Variation model proposed by Rudin et al. [19]. However, the post-processing problem is only made more computationally difficult when considering multi-valued imaging data, such as DTI or multi-channel acquisitions, wherein each voxel is a feature vector of 6-100 dimensions. In this paper, we accelerate the Total Variation regularization algorithm [15] for DTI images. To the best of our knowledge, there have been no efforts to accelerate such fundamental algorithms for DTI images in the GPGPU community before.

For this regularization algorithm to be viable for clinical settings, significant and low-cost computational acceleration is required. We find that regularization algorithms for DTI images can significantly benefit from the advances in the architecture of GPU. Solving partial differential equations in Total Variation model poses extensive synchronization on the GPU-based Implementation of TV regularization. Hence, in this paper, we analyzed the effect of synchronization by comparing our GPU-based implementation to a secondary approach that eliminates synchronization by dividing all computations into independent sub-blocks. Thereby, we compared the effect of excessive synchronization on our primary approach against the effect of excessive computational workload and memory load in the secondary approach that is imposed on each thread by eliminating the synchronization.

In particular, regularization of a tensor-valued image of dimension 128^3 completes in 1 minutes and 30 seconds, while the same regularization on a dual-core CPU requires more than 3 hours. The 128X acceleration achieved on the GPU makes this method much more appealing in clinical settings. This application study not only reveals the application-specific techniques that used for adapting this algorithm to NVIDIA G80 GPU, but also provides an insight into generally adapting Variational methods to this platform for other image processing algorithms designed for matrix-valued images.

The remainder of this paper is organized as follows. In Section 2 first describes the architecture of the Quadro FX 5600 and its G80 GPU, and then discusses the advantages of Total Variation regularization particularly for DTI images. Section 3 presents the GPU-based implementation of the TV regularization method. Section 4 describes the primary and secondary methodologies in the GPU-based implementation. Section 5 describes experimental results and compares our primary approach with the secondary approach. Section 6 discusses related work in GPU-based medical imaging. Finally, we conclude with some final statements and suggestions for future work in Section 7.

2. BACKGROUND

2.1 The Quadro FX 5600 Architecture

The NVIDIA Quadro FX 5600 is used as our main experimental platform. Quadro FX 5600 is based on G80 graphics processing unit. The Quadro has 128 processor cores with the ability to directly access a global device memory, which allows a more flexible programming model than previous generations of GPU. G80 supports the Single Instruction Multiple Data (SIMD) programming model, which is more general and flexible than the programming models supported by previous generations of GPU. This model allows data-parallel algorithms to be well suited for these architectures. NVIDIA's Compute Unified Device Architecture (CUDA) [3] was introduced by NVIDIA as a set of development tools to ease the developments on this architecture. In this section, we discuss CUDA and its threading model and the architectural features of the G80 that are most relevant to our proposed memory optimization technique. More comprehensive descriptions are found in [3, 5, 16].

CUDA Is a C-compiler that allows programmers to develop applications using a data-parallel programming model. CUDA gives developers access to the native instruction set and memory of the massively parallel computational elements of CUDA-enabled architectures. CUDA treats GPU as a coprocessor that executes data-parallel functions, so called kernel functions. The source program provided by the developer is divided into host (CPU) and kernel (GPU) code, which are then compiled by the host compiler and NVIDIA's compiler (nvcc) respectively.

2.1.1 Architectural Features

Figure 1 demonstrates Quadro's architecture. This G80 GPU consists of 16 *streaming multiprocessors* (SMs), each containing eight *streaming processors* (SPs) running at 1.35 GHz. Each SM has 8,192 registers and 16KB of on-chip *shared memory* that are shared among all threads assigned to the SM. Threads on a given SM's cores execute in SIMD fashion, with the instruction unit broadcasting the current instruction to the eight cores of the SM. Each core has a single arithmetic unit that performs double-precision floating-point arithmetic and 32-bit integer operations.

The Quadro has 76.8 GB/s of bandwidth to its off-chip *global memory.* Bandwidth to off-chip memory is quite high, but can be saturated if many threads request access within a short period of time. This bandwidth can be achieved only if accesses to the memory are contiguous 16-words lines; therefore, it is very important to follow the right access pattern to get maximum memory bandwidth. To reduce the application's demand for off-chip memory bandwidth, there are on-chip memories that can be employed to exploit the data locality and data sharing. Each SM has a 16KB of on-chip *shared memory* for data

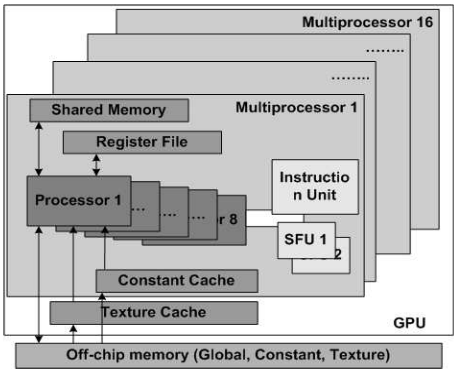


Figure 1: Architecture of Quadro FX 5600

that is either written and reused or shared among threads. In addition, Quadro has a 64 KB, off-chip *constant memory*, and each SM has an 8 KB constant memory cache. The constant memory space is cached, thus, a read from constant memory costs one memory read from global memory only on a cache miss, otherwise it just costs one read from the constant cache. Finally, for read-only data that is shared by many threads but not necessarily accessed simultaneously by all threads, the off-chip *texture memory* and the on-chip texture caches can be utilized to exploit 2D data locality to reduce the memory latency.

2.1.2 Threading Model

The batch of threads that executes the kernel in a G80 is organized as a grid of thread blocks. Each kernel creates a single grid that consists of many thread blocks. Each thread block (TB) is at most a three dimensional array of threads and has unique coordinates in the grid. Each thread block is assigned to a single SM for the duration of its execution.

The blocks that are being processed by one multiprocessor in one batch are referred to as *active*. Each active block is split into SIMD groups of threads called *warps*: each of these warps contains 32 threads. Warps are the scheduling units in SMs. All warps from all active blocks are scheduled by the thread scheduler in order to maximize the use of the multiprocessor computational resources. Therefore, SMs can interleave warps so that when one stalls on high latency memory accesses or high latency arithmetic operations, the SM switches to a ready warp in the same thread block that is assigned to the SM. The SM *occupancy* is defined as the ratio of the number of active warps per SM to the maximum number of active warps.

G80 has very limited resources available to each thread. Thus, the more resources consumed by each thread, fewer threads can be active simultaneously which results in tremendous performance loss. Therefore, there is often a trade-off between the efficiency of individual threads and thread-level parallelism. In other words, although by using more resources in each thread we can increase the performance of each thread individually, this eventually reduces the degree of parallelism, which results in reduction of the SM's occupancy [14].

2.2 Regularization of Matrix-Valued Images

Image processing methods using Variational calculus and partial differential equations (PDEs) have been popular for a long time in the image processing research community. Among popular PDE methods are the Total Variation method introduced by Rudin *et al.* [19] and various methods related to this [16, 18]. Many of these methods were extensively studied for scalar-valued (gray-scale) images and were later generalized to vector-valued (color) images. Emerging imaging modalities such as matrix-valued images, also, require robust image processing methods. However, when considering multi-valued imaging data, the computational complexity of these algorithms becomes significantly higher and makes them impractical for clinical purposes. One of the most fundamental image processing algorithms is denoising that is usually required as a pre-processing step for registration and segmentation of medical images. Thus, acceleration of such a fundamental algorithm for matrix-valued images such as DTI images has an immediate impact on medical imaging community. This paper shows that regularization of matrix-valued images becomes viable in clinical settings when accelerated on GPUs.

Emerging imaging modalities such as matrix-valued images, also, require robust image processing methods. However, when considering multi-valued imaging data, the computational complexity of these algorithms becomes significantly higher and makes them impractical for clinical purposes. One of the most fundamental image processing algorithms is denoising that is usually required as a pre-processing step for registration and segmentation of medical images. Thus, acceleration of such a fundamental algorithm for matrix-valued images such as DTI images has an immediate impact on medical imaging community. This paper shows that regularization of matrix-valued images becomes viable in clinical settings when accelerated on GPUs.

We implemented the Total Variation regularization of [15] specifically for DTI images. This algorithm finds the solution to the following minimization problem for each voxel (each voxel is a feature vector of size 6):

$$min_{l_{kl}}\{\sqrt{\underbrace{\sum_{ij} TV[d_{ij}(l_{kl})]^2}_{\mathbf{R}(u)}+\frac{1}{2}\underbrace{\sum_{ij}\parallel d_{ij}-\hat{d}_{ij}\parallel_2^2\}}_{\mathbf{F}(u,f)}} \qquad (1)$$

Where $kl \in \{11, 21, 22, 31, 32, 33\}$ and $\hat{d}_{ij}$ denotes the elements of the tensor estimated form noisy data, d_{ij} denotes the elements of matrix D and TV is the Total Variation norm of a matrix. Matrix D is defined as $D = L \cdot L^T$ where L is a lower triangular matrix. Consequently, the diffusion matrix is represented on the form of a Cholesky factorization. The objective is to find the dij as the (unique) minimizer of Eq. 1. $\mathbf{R}(u)$ is the regularization functional and $\mathbf{F}(u, f)$ is the fidelity functional. The regularization term is a geometric functional measuring smoothness of the estimated solution and the fidelity term is a measure of fitness of the estimated solution. Total Variation (TV) norm of a matrix $D \in R \times R$ is defined as:

$$TV[D] = (TV[d_{11}(l_{ij})]^2 + 2TV[d_{21}(l_{ij})]^2$$

$$+TV[d_{22}(l_{ij})]^2 + 2TV[d_{31}(l_{ij})]^2$$

$$+2TV[d_{32}(l_{ij})]^2 + TV[d_{33}(l_{ij})]^2)^{1/2} \qquad (2)$$

Eq. 3 gives the abstract formulation of the problem.

$$min_u\{\mathbf{G}(u,f,\lambda) = \mathbf{R}(u) + \frac{\lambda}{2}\mathbf{F}(u,f)\} \tag{3}$$

The minimization problem described in this paper therefore consists of five primary computations. Derivative of regularization functional **R** is given in the following equations:

$$\frac{\partial \mathbf{R}}{\partial l_{kl}} = -\sum_{ij} \frac{1}{\alpha_{ij}} \underbrace{\underbrace{TV[d_{ij}]}_{TVnorm} \underbrace{\nabla\cdot(\frac{\nabla d_{ij}}{|\nabla d_{ij}|})\frac{\partial d_{ij}}{\partial l_{kl}}}_{curvature}}_{\mathbf{P}(x_{ij})} \tag{4}$$

$$\alpha_{ij} = TV[D]$$

Throughout this paper, ∇ denotes the spatial gradient, while $\nabla\cdot$ denotes the divergence operator.

First, the algorithm computes each element of **P** as part of computing $\frac{\partial R}{\partial l_{kl}}$ given by,

$$\mathbf{P}(x_{ij}) = TV[d_{ij}]\nabla\cdot(\frac{\nabla x_{ij}}{|\nabla x_{ij}|})\frac{\partial x_{ij}}{\partial l_{kl}} \tag{5}$$

Function **P** consists of two major parts, curvature and the total variation norm. Curvature is the most computationally expensive function **P** in the algorithm. α_{ij} is the scaling factor, which scales the result of **P** based on the total variation in the image. Second, the algorithm computes the gradient of regularization functional **R**

$$\frac{\partial \mathbf{R}}{\partial l_{kl}} = -\sum_{ij}\mathbf{P}(d_{ij})\frac{\partial d_{ij}}{\partial l_{kl}} \tag{6}$$

Because all six $\frac{\partial R}{\partial l_{kl}}$ depends on values of $\mathbf{P}(d_{ij})$ the values of **P** can be computed beforehand and then reused in the computation of $\frac{\partial R}{\partial l_{kl}}$. Third, the algorithm computes the gradient of the fidelity functional **F**:

$$\frac{\partial \mathbf{F}}{\partial l_{kl}} = 2\sum_{ij}(d_{ij} - \hat{d}_{ij})\frac{\partial d_{ij}}{\partial l_{kl}} \tag{7}$$

Forth, the algorithm combines the previous computations to compute the gradient $\frac{\partial \mathbf{G}}{\partial l_{ij}}$,

$$\frac{\partial \mathbf{G}}{\partial l_{ij}} = \frac{\partial \mathbf{R}}{\partial l_{ij}} + \frac{\partial \mathbf{F}}{\partial l_{ij}}, \{ij\} \in \{11, 21, 22, 31, 32, 33\}. \tag{8}$$

Finally, the algorithm can iteratively solve the Euler-Lagrange equation corresponding to the minimization problem using the steepest descent method with a fixed time step Δt. For this step, six equations are solved iteratively based on Eq. 9.

$$d_{ij}^{n+1} = d_{ij}^{n} - \triangle t\frac{\partial \mathbf{G}^n}{\partial l_{ij}}, \{kl\} \in \{11, 21, 22, 31, 32, 33\}. \tag{9}$$

For each step of the algorithm computations are performed for six gradient directions $\{11, 21, 22, 31, 32, 33\}$, which makes each step computationally intensive. The complexity of Total Variation regularization as a denoising method for multi-valued imaging data such as MR diffusion tensor imaging or multi-channel acquisitions, far exceeds the complexity of the same methods for conventional vector-valued images, since each voxel can be a feature vector of 6-100 dimensions in multi-valued imaging. For this reason, denoising of high-resolution, three dimensional and multi-valued images have been impractical in clinical settings, despite the need for such imaging modalities. Our work demonstrates that this advanced denoising methods can be performed quickly and efficiently on modern GPUs, increasing their viability in clinical settings.

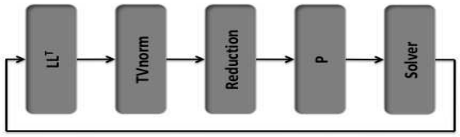


Figure 2: TV Regularization Kernel Control Flow

3. GPU-BASED IMPLEMENTATION

The Total Variation (TV) regularization method for matrix-valued images described in Section 2.2 consists of five steps in its GPU-based implementations and is generally a solver that iteratively solves a minimization problem based on steepest descent method. Each step depicted in Figure 2, can be implemented as a CUDA kernel. However, data does not need to be transferred back and forth between CPU and GPU between kernel launches, hence, avoiding the overhead. We explain the kernel control flow in the following sections.

3.1 LLT

After estimating the Cholesky factors L, tensor D is calculated per each voxel in the kernel LL^T. Kernel LL^T is a matrix multiplication kernel, multiplying a lower triangular matrix and its transpose, and is executed in data-parallel fashion. The output of kernel LL^T, $\mathbf{x}_{ij}$ is then fed to successive kernels. $\mathbf{x}_{ij}$ is a 3-D matrix, which contains the element d_{ij} of the diffusion matrix D per each voxel in the image.

3.2 TVnorm

The kernel $TVnorm$ computes the Total Variation norm for each $\mathbf{x}_{ij}$ in a data-parallel fashion, which consists of computing derivatives of the 3-D image in x, y, z directions and finally calculating the norm by performing a *sqrt* operation. Computation of $TVnorm$ is followed by a global reduction operation among all thread blocks. For this, each thread block does its own share of accumulation in the shared memory, and then a global accumulation needs to be done among the single result of all thread blocks in the global memory. However, since our GPU platform does not support atomic add at this moment , we need to add an additional kernel (kernel *Reduction*) to our design to perform the final reduction.

3.3 Reduction

This kernel launches only one thread block to perform the global reduction on the set of data captured from each thread block in kernel $TVnorm$. This step had to be added to compensate for the lack of atomic operations' support in our experimental platform.

3.4 P

$\mathbf{P}(\mathbf{x}_{ij})$ as the main function in the PDE solver is implemented as an independent kernel. Kernel P consists of two major measurements: curvature and TV norm. $TVnorm$ is implemented as an independent kernel as described in Section 3.2, and its result of will be reused in kernel P.

3.4.1 Curvature3D

The function $Curvature3D$ consists of three measurements: (1) spatial gradient of matrix $\mathbf{x}_{ij}(\nabla x_{ij})$ in the x, y, z directions, (2) gradient norm ($|\nabla x_{ij}|$) and (3) divergence of $\frac{\nabla x_{ij}}{|\nabla x_{ij}|}$ by finding its spatial gradient in the x, y, z directions and accumulating them as the result of divergence. CUDA implementation of $Curvature3D$ is depicted in Listing 1. Measurements at each step have dependency to neighboring voxels in the image. For example, measurement of spatial gradient is dependent on neighboring voxel value; measurement of gradient norm is dependent on neighboring voxels' corresponding spatial gradient; measurement of the divergence is dependent on neighboring voxels' corresponding spatial gradients and gradient norm. Therefore, all intermediate results (i.e. spatial gradient and gradient norm) need to be computed completely per neighboring voxels before proceeding to the computations in the successive steps. This requires all the threads and thread blocks to synchronize after completion of each step. However, based on the current architecture of GPUs and the insufficient synchronization primitives supported by CUDA, global synchronization of thread blocks in the GPU-based implementation of $Curvature3D$ is impossible. This is due to the fact that global synchronization involves kernel termination and creation and it is practically impossible to perform this synchronization after each computational step in the $Curvature3D$. Synchronization of threads in the same thread block is however possible but, has considerable overhead in this function.

Listing 1: Curvature3D in CUDA

```
__device__ float
Curvature3D (...)
{
   ...
if(ty <= BLOCK_3D_Y-1)
  uy[index(ty,tx,tz)] =
        (u[index(ty+1,tx,tz)]-u[index(ty,tx,tz)])/dy;

if(tx <= BLOCK_3D_X-1)
  ux[index(ty,tx,tz)] =
        (u[index(ty,tx+1,tz)]-u[index(ty,tx,tz)])/dx;

if(tz <= BLOCK_3D_Z-1)
  uz[index(ty,tx,tz)] =
        (u[index(ty,tx,tz+1)]-u[index(ty,tx,tz)])/dz;
__syncthreads();

if((ty>=1||ty<=BLOCK_3D_Y-2)&&(tx>=0||tx<=BLOCK_3D_X-2))
  Ly=(uy[index(ty,tx,tz)]+uy[index(ty,tx+1,tz)]
        +uy[index(ty+1,tx,tz)]+uy[index(ty+1,tx+1,tz)])/4;

if((tz>=1||tz<=BLOCK_3D_Z-2)&&(tz>=0||tz<BLOCK_3D_Z-))
  Lz=(uz[index(ty,tx,tz)]+uz[index(ty,tx+1,tz)]
        +uz[index(ty,tx,tz+1)]+uz[index(ty,tx+1,tz+1)])/4;

if (tx <= BLOCK_3D_X-1)
  normx[index(ty,tx,tz)] =ux[index(ty,tx,tz)]/
           (sqrtf(powf(ux[index(ty,tx,tz)],2)
           +powf(Ly,2)+powf(Lz,2)+TINY ));
__syncthreads();

if (tx <= BLOCK_3D_X-2)
    uxx=(normx[index(ty,tx+1,tz)]
                          -normx[index(ty,tx,tz)])/dx;

//NOTE:same calculations repeats as above for uyy and uzz
return ( uxx + uyy + uzz );
}
```

Listing 2: Solver Implementation in MATLAB. Results of function $\mathbf{P}(\mathbf{x}_{ij})$, gradient of R and gradient of G are computed in multiple steps and referenced as pt_{ij}, $drdl_{ij}$, $dgdl_{ij}$ respectively.

```
for i = 1:iter

D = LLT(L);
pt11=p(D(:,:,:,1,1),dx,dy,dz);
pt21=p(D(:,:,:,2,1),dx,dy,dz);
pt22=p(D(:,:,:,2,2),dx,dy,dz);

//similar calculation repeats for pt31, pt32 and pt33

drdl11=L(:,:,:,1,1).*pt11+L(:,:,:,2,1).*pt21
  +L(:,:,:,3,1).*pt31;
drdl21=L(:,:,:,1,1).*pt21+L(:,:,:,2,1).*pt22
  +L(:,:,:,3,1).*pt32;
drdl22=L(:,:,:,2,2).*pt22+L(:,:,:,3,2).*pt32;

//similar calculation repeats for drdl31,drdl32 and drdl33

dgdl11 = 2*lambda
   *(D(:,:,:,1,1) - Xnoisy(:,:,:,1,1)).*L(:,:,:,1,1)+...
   (D(:,:,:,2,1) - Xnoisy(:,:,:,2,1)).*L(:,:,:,2,1)+...
   (D(:,:,:,3,1) - Xnoisy(:,:,:,3,1)).*L(:,:,:,3,1) -...
   2*drdl11;

dgdl21 = 2*lambda
   *(D(:,:,:,2,1) - Xnoisy(:,:,:,2,1)).*L(:,:,:,1,1)+...
   (D(:,:,:,2,2) - Xnoisy(:,:,:,2,2)).*L(:,:,:,2,1)+...
   (D(:,:,:,3,2) - Xnoisy(:,:,:,3,2)).*L(:,:,:,3,1) -...
   2*drdl21;

dgdl22 = 2*lambda
   *(D(:,:,:,2,2) - Xnoisy(:,:,:,2,2)).*L(:,:,:,2,2)+...
    (D(:,:,:,3,2) - Xnoisy(:,:,:,3,2)).*L(:,:,:,3,2) -...
    2*drdl22;

//similar calculation repeats for dgdl31,dgdl32 and dgdl33

L(:,:,:,1,1)=L(:,:,:,1,1) - dt*dgdl11;
L(:,:,:,2,1)=L(:,:,:,2,1) - dt*dgdl21;
L(:,:,:,2,2)=L(:,:,:,2,2) - dt*dgdl22;

//similar calculation repeats for 31,32 and 33
end
```

The GPU-based implementation of the $Curvature3D$ uses shared memory to ameliorate the effect of this global synchronization problem by overlapping the boundaries among thread blocks and creating redundant computations in a given block of image. As a result, for a $N \times N \times N$ data block, we create a thread block with size $(N + 2) \times (N + 2) \times (N + 2)$. After completing each computational step (spatial gradient, gradient norm and divergence) in $Curvature3D$ the dimension of active threads is decreased by one. Therefore, at the beginning of the kernel P, there exists $(N + 2) \times (N + 2) \times (N + 2)$ active threads for loading a data block of size $(N + 2) \times (N + 2) \times (N + 2)$, while at the end there exist only $N \times N \times N$ active threads.

3.5 Solver

Kernel *Solver* is a data-parallel implementation of the PDE solver in the TV regularization algorithm. As described in Section 2.2, Eq. 1 is solved iteratively by invoking this kernel to find the unique minimizer of the problem. The kernel is invoked until the number of iterations exceeds a threshold. At each iteration, the solver finds the gradient of $\mathbf{G}$ per voxel in multiple steps as described in Eq. 4-8 to solve the Euler-Lagrange equation given by Eq. 9. Results of kernel P stored in global memory is used in this kernel for computing the gradient of $\mathbf{R}$ based on Eq. 4. Finally, $\mathbf{x}_{ij}$ is estimated according to the Euler-Lagrange equation given by Eq. 9, based on gradient of $\mathbf{G}$ and the value of $\mathbf{x}_{ij}$ in the previous iteration. The implementation for the solver is demonstrated in MATLAB for the sake of briefness in Listing 2, which consists of function $\mathbf{P}$ and LLT which are implemented as separate kernels in the CUDA implementation.

4. METHODOLOGY

Regularization of multi-valued images using the algorithm described in Section 2.2 imposes significant synchronization overhead to its GPU-based implementation resulting from the structure of the algorithm especially function $\mathbf{P}(\mathbf{x}_{ij})$, which is the most computationally expensive function in this application. We are particularly aimed at evaluating the effects of excessive synchronization in this application study. The synchronization overhead in the $Curvature3D$ is an interesting behavior to study in the GPGPU domain. Therefore, it is worth comparing several approaches in implementing synchronization in the $Curvature3D$ for the sake of performance evaluations in order to learn the effect of synchronization in similar applications. This property of $Curvature3D$ is in contrast to most of the previous GPGPU applications that are successfully ported to GPUs [20, 21]. Therefore, in addition to *GPU.GlobalSync* as our primary GPU-based implementation, secondary approaches to our GPU-based implementation are demonstrated in this paper. We consider our secondary approaches as approximate GPU-based solutions of the TV Regularization algorithm, and only present them for the sake of performance evaluations of different synchronization patterns. Therefore, they are not fully optimized. Function $Curvature3D$ in kernel P is particularly important for our evaluation; therefore, we focus particularly on this function.

4.1 Primary Approach

This GPU-based implementation (*GPU.GlobalSync*) executes in data parallel fashion on the GPU. In this layout, each thread is responsible for computations in a single voxel. Kernel P is the most computationally intensive among other kernels. Computing the intermediate results in $Curvature3D$ function within kernel P (i.e. spatial gradient, gradient norm, and divergence corresponding to each voxel) has substantial data reuse among threads within a thread block, therefore, placing them in shared memory hides the excessive memory latencies. Moreover, coordination among all the threads and thread blocks is necessary for consistency. In *GPU.GlobalSync*, global synchronization among thread blocks is achieved by overlapping the boundaries among thread blocks and redundant computation in a given block of image as elaborated in Section 1. For example, for each 6×6 data block in the image we actually create a 8×8 thread block and load 8×8 data blocks to shared memory. On the other hand, thread-level synchronization is enforced by CUDA provided synchronization primitive among threads in the thread block. Other kernels, however, do not require the same layout in terms of the work distribution among threads since no specific coordination is required among the threads nor thread blocks. All computations in kernels that will be used in successive kernels are stored in global memory, where all accesses to off-chip memory are coalesced to conserve its bandwidth.

4.2 Secondary Approach

In both of our secondary approaches, since we are not mainly concerned about detailed analysis of these implementations and mainly focused on evaluating the effect of synchronization, we did not perform extensive performance tuning for our secondary implementations. Moreover, we do not follow the same kernel composition as *GPU.GlobalSync* described in Section 3. Hence, we consider only one kernel, which we refer to as Solver, and the main device functions are: function $\mathbf{P}$ calling $Curvature3D$ and $TVnorm$, and function LLT. Both secondary approaches are approximate in the sense that the TV norm is only computed only for each thread block and therefore, no global reduction is required which eliminates the need for the *Reduction* kernel in this layout.

4.2.1 Eliminating Synchronization

In this GPU-based implementation ($GPU.UnSync$), we relax the synchronization problem in the TV regularization algorithm that existed in $Curvature3D$. In order to alleviate this synchronization problem, all computations are divided into independent sub-blocks (i.e. cubes for 3-D images) or sub-matrices in the image. Thus, in $GPU.UnSync$ each thread is responsible for computations in a sub-block in contrast to *GPU.GlobalSync* where each thread is responsible for computations in a single voxel. In our implementation, the size of each sub-block is $2 \times 2 \times 2$. By dividing the regularization tasks into sub-blocks in the image, each sub-block (sub-image) is denoised (regularized) independent from neighboring sub-blocks, which eliminates dependency to neighboring voxels. However, the downside of $GPU.UnSync$ is the decreased quality of the denoised image compared to the original algorithm that performs denoising at a global image-level. This approach enforces data-parallelism at a higher granularity than *GPU.GlobalSync* and eliminates the need for synchronization between neighboring voxels within the sub-block. However, dealing with the boundary data between threads is still a remaining challenge. We enforce padding the boundary of sub-blocks that eliminates the need to exchange boundary data between threads, and therefore, we can relax the synchronization problem between threads. Because of the high memory load in this implementation, the amount of off-chip memory latencies that can be hidden by leveraging the hardware's data transfer mechanism is limited here, because constant and texture memories are both read-only and shared memory is very limited to fit all the intermediate results. However, in order to conserve bandwidth to off-chip memory, memory coalescing is enabled as much as possible.

4.2.2 Thread-level Synchronization

This GPU-based implementation ($GPU.ThreadSync$) executes in data parallel fashion on the GPU. In this layout, each thread is responsible for computations in a single voxel the same as our primary (*GPU.GlobalSync*) implementation. Thread-level synchronization is enforced by CUDA provided synchronization primitive among threads in the thread block. On the other hand, global synchronization is still not possible in this layout. In this layout, all the intermediate results in $Curvature3D$ (i.e. spatial gradient and gradient norm, etc corresponding to each voxel) are stored in global memory as opposed to each thread's local memory since each thread needs to access the intermediate results computed by neighboring threads in the thread block. For the sake of performance comparison of $GPU.ThreadSync$ with $GPU.UnSync$, no hardware's data transfer mechanism (e.g. utilizing shared memory) is leveraged in this layout to be consistent with *GPU.UnSync*; since we only want to evaluate the effect of synchronization in this study; therefore, we don't want a better memory placement option dramatically changes the result in the favor of *GPU.ThreadSync*. However, in order to conserve bandwidth to off-chip memory, memory coalescing is enabled as much as possible. Boundary data in the *GPU.ThreadSync* implementation is handled by padding the boundary of thread blocks' corresponding region that eliminates the need to exchange boundary data between thread blocks. The Kernel Solver on the other hand, exactly follows the same layout as the kernel Solver in our primary GPU-based implementations.

5. EXPERIMENTAL EVALUATION

This section presents the experimental results of accelerating the TV regularization algorithm on GPUs. In this section, we analyze different characteristics of our primary and secondary implementations. We used CUDA version 2.0 for our GPU-based implementations. Experiments were performed on Intel Core2 Duo running at 2.33 GHz with 8GB of main memory and NVIDIA Quadro FX 5600 as a commodity GPU. The CPU code is compiled under GCC with the $-$O3 flag. Given the same input data set, the speedup is calculated by taking the wall-clock time required by the application on the CPU divided by the time required by the GPU. Times are measured after initial setup and do not include PCI-E bus transfer time. In this section we mainly analyze the *Solver* as the main computational kernel.

5.1 Primary Approach

As Figure 3 shows, *GPU.GlobalSync* achieves up to 266X speedup over the CPU version. In this version, there are 256 threads per thread block and each grid processes 128^3 matrix-valued tensors. In Kernel P, each thread uses 10 registers, and shared memory usage is 6 KB per thread block. Therefore, up to 2 thread blocks can execute on each SM simultaneously, which represents 66% utilization of the Quadro's processor cores. In Kernel $TVnorm$, each thread uses 10 registers, and shared memory usage is 2 KB per thread block. Therefore, up to 3 thread blocks can execute on each SM simultaneously, which represents 100% utilization of the Quadro's processor cores. Kernel *Solver* uses 27 registers per thread, and therefore, up to 1 thread blocks can execute on each SM simultaneously, which represents 33% utilization. Kernel *Solver* has high off-chip memory load. The ratio of floating-point operations to memory accesses is 0.84; therefore, knowing that the memory bandwidth is 76.8 GB/s, the upper limit on kernel *Solver* performance is only 16.12 GFLOPS.

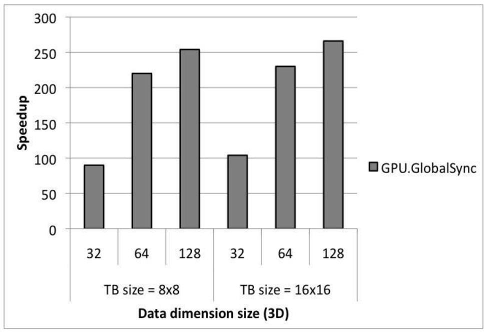


Figure 3: Kernel Speedup for Our Primary GPU-based Implementation

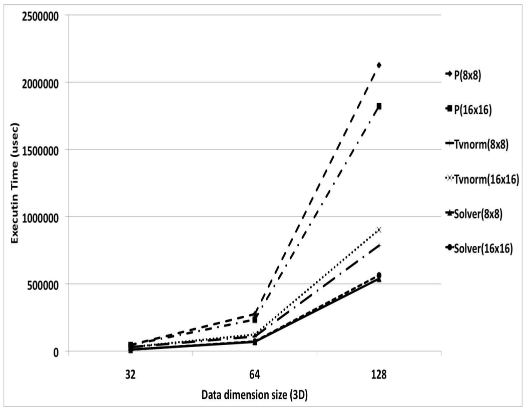


Figure 4: Kernel Execution time for Our Primary GPU-based Implementation

Table 1 demonstrates properties of each kernel. Kernel P and kernel $TVnorm$ constitute the highest percentage of the total execution time. Kernel's execution time include the kernel creation and termination overhead. It is observed that the highest overhead belongs to kernel *Reduction*, which only occupies one multiprocessor in the device. Kernel *Solver* has high usage of registers, therefore, in spite of high kernel creation and termination overhead it is essential to decompose the computation of the PDE solver into multiple kernels in this layout. Increasing the TB size from 8 × 8 to 16 × 16 does not have significant effect on overall performance as demonstrated in Figure 3. However, it is observed from Figure 4 that the increase in TB size, decreases the execution time in kernel P. This observation shows that current GPU architectures has tolerance for thread-level synchronization. On the other hand, increase in TB size increases the execution time in kernel $TVnorm$. These observations are most evident in large data sizes. Therefore, the reverse effect of TB size increase on both kernels has somewhat diminished the effect of increasing TB size on overall performance.

5.2 Secondary Approach

As Figure 5 shows, *GPU.ThreadSync* achieves up to 128X speedup over the CPU version. In this version, there are 64 threads per block and each grid processes 128^3 matrix-valued tensors. Each thread uses 30 registers, and therefore, up to 8192/30=273 threads can execute on each SM simultaneously, which represents 33% utilization of the Quadro's processor cores. *GPU.UnSync* on the other hand achieves up to 134X speedup over the CPU version as depicted in Figure 5. It is notable that *GPU.ThreadSync* achieves higher speedup compare to *GPU.UnSync* with data size of 128^3 when TB size is 8 × 8. Furthermore, *GPU.ThreadSync* shows to scale better with the increase in data size. Figure 6 depicts the execution time of *GPU.UNSync* compared to *GPU.ThreadSync*. As it is observed from Figure 6(b) when TB size is increased to 16 × 16, by increasing the data size to 128^3 in *GPU.ThreadSync*, the growth in execution time has a

Table 1: Kernel Implementation Performance for Execution Profiles

Kernel	#Calls	Execution time (ms)	Utilization	%GPU Time	Shared Mem per Thread Block (KB)	Registers per Thread
LLT	100	258.13	100%	7%	0	9
TVnorm	600	888.21	100%	21%	2	10
Reduction	100	178.19	66%	2.86%	2	4
P	600	1800.00	66%	54.14%	6	10
Solver	600	557.79	33%	13.03%	0	27

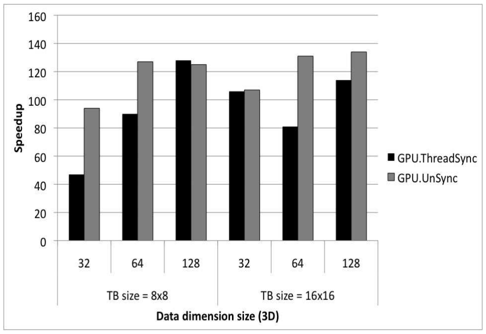


Figure 5: Kernel Speedup for Our Secondary GPU-based Implementations

slower slope than that of *GPU.UnSync*. This demonstrates that GPU architecture has better tolerance for excessive synchronization in *GPU.ThreadSync* rather than excessive per thread computational workload and memory load that exist in *GPU.UnSync*. That is due to the fact that, in *GPU.UnSync*, all computations in each thread are performed on 2 × 2 × 2 blocks of data as opposed to one single voxel in *GPU.ThreadSync*.

This is made clearer in Figure 7, where the gap between execution time of function **P** and the overall execution time of kernel Solver is more evident in *GPU.UnSync* compared to *GPU.ThreadSync* (computation of **P** is part of Solver in secondary implementations). Although execution of **P** is faster in *GPU.UnSync*, the remaining computations of Solver, takes more time to complete compared to *GPU.ThreadSync* duo to excessive computational workload and memory load of each thread in *GPU.UnSync*. Therefore, the negative effect of excessive synchronization in *GPU.ThreadSync* is made less evident. Overall, it is notable that the speedup achieved from *GPU.UnSync* is not considerably better than *GPU.ThreadSync* in large data sizes.

6. RELATED WORK

The challenges in the GPGPU community have revolved around the constraints of the programming environment and on optimal mapping of applications so to best leverage the highly parallel GPU architecture. There have been several attempts in acceleration of scientific, imaging, database management and data mining [23, 21]. Medical imaging was one of the first GPU computing applications with acceleration of CT reconstruction [24]. The acceleration of an advanced MRI reconstruction algorithm is also studied in [20]. Currently a lot of work is done with Variational methods, that become real-time applicable by implementing them on the GPU. [25] discusses fast and accurate methods to solve Total Variation models on GPUs. In their work, two prominent models incorporating TV regularization are reviewed and two different algorithms are presented to solve these models. Moreover, a demonstration of Variational methods for diverse computer vision tasks such as registration, segmentation and stereo matching using GPUs is given in [26]. Research in this area has mainly focused on Total Variation methods for scalar-valued (gray-scale) and vector-valued (color) images. By contrast, Variational methods for matrix-valued imaging data particularly DTI has not been studied before in the GPGPU domain.

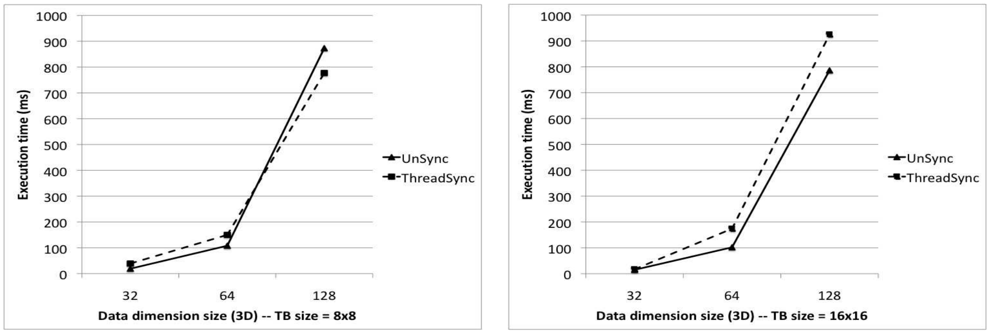
7. CONCLUSIONS

Multi-valued imaging such as diffusion tensor imaging (DTI) has substantially higher noise levels compared to conventional MR imaging. Total Variation regularization, which is particularly designed for DTI images, can mitigate these limitations at the expense of substantial computation. The regularization algorithms for DTI images can significantly benefit from the advances in the architecture of GPU and reduce the execution time of regularization of matrix-valued images from 3 hours in a single-threaded implementation on a dual-core CPU to 1 minutes and 30 seconds, making the application of DTI images practical for many clinical settings.

We analyzed the effect of excessive synchronization in this algorithm, which results from dependence of this method to solving partial differential equations. The observations from this study showed that current GPU architectures has tolerance for excessive synchronization. We further analyzed the effect of synchronization by comparing our two secondary implementations. Thereby, we compared the effect of excessive synchronization against the effect of excessive computational workload and memory load that is imposed on each thread by eliminating the synchronization. This study showed that although the approach in which thread-level synchronization is eliminated achieves higher speedup, but the approach which retains thread-level synchronization scales better on the Quadro by the increase in image size.

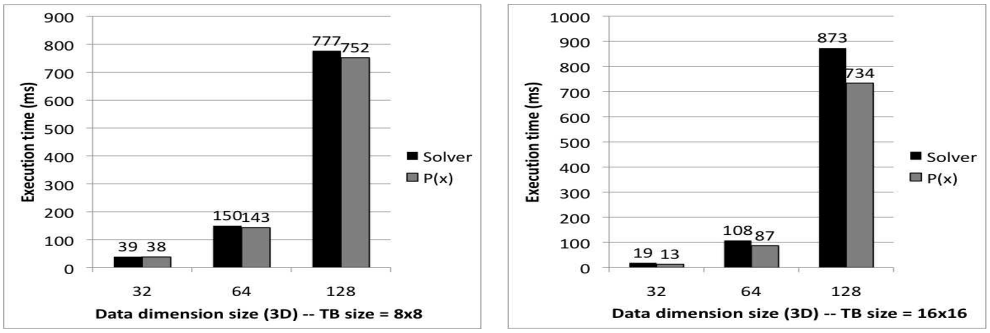
8. ACKNOWLEDGMENTS

The authors wish to thank Tin Man Lee from UCLA



(a) (b)

Figure 6: GPU-based Solver Execution Time



(a) (b)

Figure 7: GPU-based Solver Execution Time

Biomedical Engineering Department for providing the initial code for this application in MATLAB. This material is based on work supported under NLM Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NIH. Experiments were made possible by generous donations of hardware from NVIDIA.

9. REFERENCES

[1] D. Manocha, M. C. Lin, N. Govindaraju. GPGPU to Many-Core Processing: Higher Performance for Mass Market Applications. Manycore Computing Workshop, 2007.

[2] NVIDIA Corporation. NVIDIA CUDA Programming Guide, version 1.1, 2007.

[3] AMD Stream Processor. http://ati.amd.com/products/streamprocessor/index.html.

[4] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 2.0). Silicon Graphics, Inc., October 2004.

[5] DirectX Developer Center. http://www.msdn.com/directx/.

[6] Cg. http://developer.nvidia.com/page/cg main.html.

[7] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In Proceedings of the 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 325-335, 2006.

[8] I. Buck. Brook Specification v0.2, October 2003.

[9] P. J. Basser, J. Mattiello, and D. LeBihan, MR diffusion tensor spectroscopy and imaging, Biophysical Journal, vol. 66, no. 1, pp. 259-267, 1994.

[10] D. Le Bihan, J.-F. Mangin, C. Poupon, et al., Diffusion tensor imaging: concepts and applications, Journal of Magnetic Resonance Imaging, vol. 13, no. 4, pp. 534-546, 2001.

[11] C.-F. Westin, S. E. Maier, H. Mamata, A. Nabavi, F. A. Jolesz, and R. Kikinis, Processing and visualization for diffusion tensor MRI, Medical Image Analysis, vol. 6, no. 2, pp. 93-108, 2002.

[12] S. Mori and P. B. Barker, Diffusion magnetic resonance imaging: its principle and applications, The Anatomical Record vol. 257, no. 3, pp. 102-109, 1999.

[13] S. Mori and P. C. M. van Zijl, Fiber tracking: principles and strategies - a technical review, NMR in Biomedicine, vol. 15, no. 7-8, pp. 468-480, 2002.

[14] R. Bammer, Basic principles of diffusion-weighted imaging, European Journal of Radiology, vol. 45, no. 3, pp. 169-184, 2003.

[15] O. Christiansen, T. M. Lee, J. Lie, U. Sinha, and T. F. Chan, Total Variation Regularization of Matrix-Valued Images, International Journal of Biomedical Imaging, vol. 2007, 11 pages, 2007.

[16] M. Lysaker, S. Osher, and X.-C. Tai, Noise removal using smoothed normals and surface fitting, IEEE Transactions on Image Processing, vol. 13, no. 10, pp. 1345-1357, 2004.

[17] T. F. Chan and S. Esedoglu, Aspects of total variation regularized L1 function approximation, SIAM Journal on Applied Mathematics, vol. 65, no. 10, pp. 1345-1357, 2005.

[18] J. Weickert and T. Brox, Diffusion and regularization of vector- and matrix-valued images, Tech. Rep. preprint no. 58, Fachrichtung 6.1 Mathematik, Universitat des Saarlandes, Saarbr ucken, Germany, 2002.

[19] L. I. Rudin, S. Osher, and E. Fatemi, Nonlinear total variation based noise removal algorithms, Physica D, vol. 60, no. 1-4, pp. 259-268, 1992.

[20] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, and W. Hwu. Program optimization study on a 128-core GPU. First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), 2007.

[21] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, Z. P. Liang, B. P. Sutton, Accelerating Advanced MRI Reconstructions on GPUs, Proceedings of the 2008 International Conference on Computing Frontiers, May 2008.

[22] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. 68, 10 (Oct. 2008), 1370-1380.

[23] J. Owens, D. Luebke, N. Govindara ju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26(1):80-113, March 2007.

[24] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In 1994 Symposium on Volume Visualization, 1994.

[25] Pock, T.; Unger, M.; Cremers, D.; Bischof, H., Fast and exact solution of Total Variation models on the GPU, Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on , vol., no., pp.1-8, 23-28 June 2008.

[26] T. Pock, M. Grabner, and H. Bischof. Real-time Computation of Variational Methods on Graphics Hardware, Computer Vision Winter Workshop 2007, Michael Grabner, Helmut Grabner, St. Lambrecht, Austria, February 6-8.