GPU-ACCELERERATED REGULARISATION OF LARGE DIFFUSION-TENSOR VOLUMES

TUOMO VALKONEN† AND MANFRED LIEBMANN†

Abstract. We discuss the benefits, difficulties, and performance of a GPU implementation of the Chambolle-Pock algorithm for TGV (total generalised variation) denoising of medical diffusion tensor images. Whereas we have previously studied the denoising of 2D slices of 2×2 and 3×3 tensors, attaining satisfactory performance on a normal CPU, here we concentrate on full 3D volumes of data, where each 3D voxel consists of a symmetric 3×3 tensor. One of the major computational bottle-necks in the Chambolle-Pock algorithm for these problems is that on each iteration at each voxel of the data set, a tensor potentially needs to be projected to the positive semi-definite cone. This in practise demands the QR algorithm, as explicit solutions are not numerically stable. For a $128 \times 128 \times 128$ data set, for example, the count is 2 megavoxels, which lends itself to massively parallel GPU implementation. Further performance enhancements are obtained by parallelising basic arithmetic operations and differentiation. Since we use the relatively recent OpenACC standard for the GPU implementation, the article includes a study and critique of its applicability.

1. Introduction

Let $f \in L^1(\Omega; \mathrm{Sym}^2(\mathbb{R}^d))$ be a second-order symmetric tensor field on a domain $\Omega \subset \mathbb{R}^d$, such as a medical diffusion tensor image (DTI). In the following, we study the numerical solution of the denoising problem

$$\min_{u \geq 0} \frac{1}{2}\|f-u\|_{F,2}^2 + \mathrm{TGV}^2_{(\beta,\alpha)}(u). \tag{1}$$

Here TGV^2 denotes the second-order total generalised variation (TGV^2), introduced in [5], and extended to the tensor case in [23]. The advantage of TGV^2 over total variation (TV) is that it tends to avoid the stair-casing effect. [6, 15] Observe also that we require the solution $u \in L^1(\Omega; \mathrm{Sym}^2(\mathbb{R}^d))$ of (1) to be pointwise positive semi-definite: non-positive definite diffusion tensors are non-physical.

Diffusion-tensor images (DTI) arise from combination of multiple magnetic resonance images (MRI) obtained with different diffusion-sensitising gradients (DWI), as described by the Stejskal-Tanner equation. [4, 14]. They describe the anisotropic diffusion of water molecules, and provides valuable insight into the white matter structure of the brain [4, 20], as white matter has high anisotropy, in contrast to low-anisotropy grey matter. Diffusion-weighted MRI has long acquisition times, even with ultra fast sequences like echo planar imaging (EPI). Therefore, it is inherently a low-resolution and low-SNR method, exhibiting Rician noise [12], eddy-current distortions [20]. Additionally, it is very sensitive to patient motion. [13, 1]

Since the DWI measurements are noisy, we are led to the problem of denoising the diffusion tensors obtained obtained this way. Here we concentrate on the denoising model (1). Other approaches (see, e.g., [3, 17, 21]) are discussed and

†Institute for Mathematics and Scientific Computing, University of Graz, Graz, Austria

E-mail addresses: tuomo.valkonen@iki.fi, manfred.liebmann@uni-graz.at.

compared in our earlier work. [23] It should be pointed out that, strictly speaking, the omnipresent squared fidelity function also employed in (1) introduces the incorrect (Gaussian) noise model: As discussed out above, the DWI data has Rician distribution, and the tensor field f is linearly related to the log-ratios of the DWI images through Stejskal-Tanner equation. The squared fidelity is clearly computationally significantly easier, although potentially improved denoising results could be obtained by more correct modelling.

We choose to solve the numerical problem (1) using the Chambolle-Pock algorithm [9], as it tends to be quite efficient and easy to implement, considering the non-smoothness of (1). Because of the positive semi-definiteness constraint, however, the algorithm will involve projections to the positive semi-definite cone. These are computationally intensive, and lead to long computational times on a lone traditional CPU when denoising full 3D volumes instead of individual 2D slices. Since these projections, along with many other operations in the algorithm, are fully parallel over voxels, graphics processors should be the ideal hardware. In this paper, we study what kind of performance advantage GPUs have over CPU, as well as the implementation details and hurdles.

We have chosen to implement our GPU realisation of the code using the recent OpenACC standard instead of writing CUDA code. This consists of **#pragma** directives in relatively standard C code, and instruct an accelerator compiler (such as PGI PGCC) to generate the GPU code. As such, our study also provides a study of the practicability of OpenACC and PGCC: promising, but not fully satisfactory.

The rest of this paper is organised as follows. In Section 2 we introduce the tensor and tensor field calculus to set up the framework in which our results are represented. Then in Section 3 we formulate the problem (1) in detail. Section 4 describes the Chambolle-Pock algorithm [9] that we use to solve these problems. Then, in Section 5 we study GPU performance for this algorithmic solution. We finish the paper with our conclusions in Section 6.

2. Tensors and tensor fields

We now recall basic tensor calculus, as needed for the development of TGV^2. We make many simplifications, as we do not need the calculus in its full differential-geometric setting, working on the Euclidean space $\mathbb{R}^m$.

Basic tensor calculus. A k-tensor $A \in \mathcal{T}^k(\mathbb{R}^m)$ is a k-linear mapping $A : \mathbb{R}^m \times \cdots \times \mathbb{R}^m \to \mathbb{R}$. A symmetric tensor $A \in \mathrm{Sym}^k(\mathbb{R}^m)$ satisfies for any permutation π of $\{1, \ldots, k\}$ that $A(c_{\pi 1}, \ldots, c_{\pi k}) = A(c_1, \ldots, c_k)$.

For example, vectors $A \in \mathbb{R}^m$ can be identified with symmetric 1-tensors, $A(x) = \langle A, x\rangle$, while matrices can be identified with 2-tensors, $A(x, y) = \langle Ax, y\rangle$. Symmetric matrices $A = A^T$ can be identified with symmetric 2-tensors. We use the notation $A \geq 0$ for positive semi-definite A.

Let now $e_1, \ldots, e_m$ be the standard basis of $\mathbb{R}^m$. We then define the inner product

$$\langle A, B\rangle := \sum_{p\in\{1,\ldots,m\}^k} A(e_{p_1}, \ldots, e_{p_k})B(e_{p_1}, \ldots, e_{p_k}),$$

and the Frobenius norm

$$\|A\|_F := \sqrt{\langle A, A\rangle}.$$

For example, for 1-tensors, i.e., vectors, the inner product is the usual inner product in $\mathbb{R}^m$, and the Frobenius norm $\|A\|_F = \|A\|_2$. For 2-tensors, i.e., matrices, the inner product is $\langle A, B\rangle = \sum_{i,j} A_{ij}B_{ij}$ and $\|A\|_F$ is the matrix Frobenius norm.

Tensor fields. Let $u : \Omega \to \mathcal{T}^k(\mathbb{R}^m)$ for a domain $\Omega \subset \mathbb{R}^m$. We then set

$$\|u\|_{F,p} := \Big(\int_\Omega \|u(x)\|_F^p \, dx\Big)^{1/p} \quad (p \in [1, \infty)),$$

and define the spaces

$$L^p(\Omega; \mathcal{T}^k(\mathbb{R}^m)) = \{u : \Omega \to \mathcal{T}^k(\mathbb{R}^m) \mid \|u\|_{F,p} < \infty\}, \quad \text{and}$$
$$L^p(\Omega; \mathrm{Sym}^k(\mathbb{R}^m)) = \{u : \Omega \to \mathrm{Sym}^k(\mathbb{R}^m) \mid \|u\|_{F,p} < \infty\}, \quad (p \in [1, \infty)).$$

Finally, for $u \in C^1(\Omega; \mathcal{T}^k(\mathbb{R}^m))$, $k \geq 1$, the divergence $\operatorname{div} u \in C(\Omega; \mathcal{T}^{k-1}(\mathbb{R}^m)$ is defined by contraction as

$$\begin{aligned}[\operatorname{div} u(x)](e_{i_2}, \ldots, e_{i_k}) &:= \sum_{i_1=1}^{m} \partial_{i_1}[x \mapsto u(x)(e_{i_1}, \ldots, e_{i_k})] \\ &= \sum_{i_1=1}^{m} \langle e_{i_1}, \nabla u(\cdot)(e_{i_1}, \ldots, e_{i_k})\rangle.\end{aligned} \tag{2}$$

Observe that if u is symmetric, then so is $\operatorname{div} u$.

Let, for example, $u \in C^1(\Omega; \mathbb{R}^m) = C^1(\Omega; \mathcal{T}^1(\mathbb{R}^m))$. Then the divergence $\operatorname{div} u(x) = \sum_{i=1}^m \partial_i u_i(x)$ is the usual vector field divergence. If, on the other hand, $u \in C^1(\Omega; \mathcal{T}^2(\mathbb{R}^m))$, then $[\operatorname{div} u(x)]_j = \sum_{i=1}^m \partial_i u_{ij}(x)$. That is, we take columnwise the divergence of a vector field. For 2-tensor fields, we use the notation $u \geq 0$ for pointwise a.e. positive semi-definite u.

Denoting by X^* the continuous linear functionals on the topological space X, we now define the symmetrised distributional gradient

$$Eu \in [C_c^\infty(\Omega; \mathrm{Sym}^{k+1}(\mathbb{R}^m))]^*$$

by

$$Eu(\varphi) := -\int_\Omega \langle u(x), \operatorname{div} \varphi(x)\rangle \, dx, \quad (\varphi \in C_c^\infty(\Omega; \mathrm{Sym}^{k+1}(\mathbb{R}^m))).$$

Let us also define the "symmetric Frobenius unit ball"

$$V_{F,\mathrm{s}}^k := \{\varphi \in C_c^\infty(\Omega; \mathrm{Sym}^k(\mathbb{R}^m)) \mid \|\varphi\|_{F,\infty} \leq 1\}.$$

If $\sup\{Eu(\varphi) \mid \varphi \in V_{F,\mathrm{s}}^{k+1}\} < \infty$, then Eu is a measure [10, §4.1.5]. Indeed, for our purposes it suffices to define a tensor-valued measure $\mu \in \mathcal{M}(\Omega; \mathrm{Sym}^k(\mathbb{R}^m))$ to as a linear functional $\mu \in [C_c^\infty(\Omega; \mathrm{Sym}^k(\mathbb{R}^m))]^*$ that is bounded in the sense that the total variation norm

$$\|\mu\|_{F,\mathcal{M}(\Omega;\mathrm{Sym}^k(\mathbb{R}^m))} := \sup\{\mu(\varphi) \mid \varphi \in V_{F,\mathrm{s}}^k\} < \infty.$$

For smooth 1-tensor fields $u \in C^\infty(\Omega; \mathbb{R}^m)$, we get $Eu(x) = (\nabla u(x) + (\nabla u(x))^T/2$. Those $u \in L^1(\Omega; \mathbb{R}^m)$ for which Eu is a measure, are called *functions of bounded deformation* [19]. For scalar fields $u \in L^1(\Omega)$, the symmetrised gradient is the usual gradient, $Eu = Du$.

3. Total generalised variation of tensor fields

We now develop second-order total generalised variation for tensor fields. We begin by recalling total variation of scalar fields.

For recollection: TV and ROF for scalar fields. Let $\Omega \subset \mathbb{R}^m$ be a domain and $u \in L^1(\Omega)$. We write the *total variation* of u simply as

$$\mathrm{TV}(u) := \sup_{\varphi \in V_{F,\mathrm{s}}^1} \int_\Omega u(x) \operatorname{div} \varphi(x) \, dx = \|Du\|_{\mathcal{M}(\Omega)},$$

Given a regularisation parameter $\alpha > 0$, the ROF regularisation of $f \in L^1(\Omega)$ is then given by the solution $\hat{u}$ of the problem

$$\min_{u \in \mathrm{BV}(\Omega)} \frac{1}{2}\|f - u\|_{L^2(\Omega)}^2 + \alpha \mathrm{TV}(u).$$

Second-order total generalised variation (TGV^2) for tensor fields. Total generalised variation was introduced in [5] as a higher-order extension of TV, that tends to avoid the stair-casing effect. Given parameters $\alpha, \beta > 0$, for a scalar field $u \in L^1(\Omega)$, second-order TGV may according to [6, 7] be written as the "differentiation cascade"

$$\mathrm{TGV}^2_{(\beta,\alpha)}(u) := \min_{w \in L^1(\Omega;\mathrm{Sym}^1(\mathbb{R}^m))} \alpha \|Eu - w\|_{F,\mathcal{M}(\Omega;\mathrm{Sym}^1(\mathbb{R}^m))} + \beta \|Ew\|_{F,\mathcal{M}(\Omega;\mathrm{Sym}^2(\mathbb{R}^m))}. \tag{3}$$

Readily this definition extends to tensor fields $u \in L^1(\Omega; \mathrm{Sym}^k(\mathbb{R}^m))$ by raising the order of all the involved tensors, so that

$$\mathrm{TGV}^2_{(\beta,\alpha)}(u) := \min_{w \in L^1(\Omega;\mathrm{Sym}^{k+1}(\mathbb{R}^m))} \alpha \|Eu - w\|_{F,\mathcal{M}(\Omega;\mathrm{Sym}^{k+1}(\mathbb{R}^m))} + \beta \|Ew\|_{F,\mathcal{M}(\Omega;\mathrm{Sym}^{k+2}(\mathbb{R}^m))}. \tag{4}$$

In the rest of this paper, given a tensor field $f \in L^1(\Omega; \mathrm{Sym}^2(\mathbb{R}^m))$, we study the numerical solution of the positivity-constrained TGV^2 regularisation problem

$$\min_{0 \le u \in L^1(\Omega;\mathrm{Sym}^2(\mathbb{R}^m))} \frac{1}{2}\|f - u\|^2_{F,2} + \mathrm{TGV}^2_{(\beta,\alpha)}(u). \tag{P-TGV^2}$$

For suitable choice of parameters $(\beta, \alpha) > 0$, solutions u of this problem should then closely match the original measurement f, however missing unwanted noise, and not violating the positivity constraint that diffusion tensor fields should satisfy.

For the numerical solution, we write (P-TGV^2) in an alternative way, as a min-sup problem. Denoting by

$$\delta_A(x) := \begin{cases} 0, & x \in A, \\ \infty, & x \notin A, \end{cases}$$

the indicator function of a set A in the sense of convex analysis, and particularly by

$$\delta_{\ge 0}(u) := \begin{cases} 0, & u(x) \text{ is positive semi-definite for a.e. } x \in \Omega, \\ \infty, & \text{otherwise,} \end{cases},$$

the problem (P-TGV^2) can be written in the form

$$\min_v \sup_\xi G(v) + \langle v, K^*\xi \rangle - F^*(\xi). \tag{S-TGV^2}$$

Here the primal variables $v = (u, w)$, and dual variable $\xi = (\varphi, \psi)$ with

$$\begin{aligned} &u \in L^1(\Omega; \mathrm{Sym}^2(\mathbb{R}^m)), && w \in L^1(\Omega; \mathrm{Sym}^3(\mathbb{R}^m)), \\ &\varphi \in C^\infty_c(\Omega; \mathrm{Sym}^3(\mathbb{R}^m)), && \psi \in C^\infty_c(\Omega; \mathrm{Sym}^4(\mathbb{R}^m)). \end{aligned}$$

The functionals G and F^* and the operator K^* are defined as

$$\begin{aligned} K^*(\varphi, \psi) &:= (-\operatorname{div}\varphi, -\varphi - \operatorname{div}\psi), & G(u, w) &:= \frac{1}{2}\|f - u\|^2_{F,2} + \delta_{\ge 0}(u), \\ & & F^*(\varphi, \psi) &:= \delta_{\alpha V^3_{F,s}}(\varphi) + \delta_{\beta V^4_{F,s}}(\psi). \end{aligned}$$

Observe that we bound φ pointwise by the Frobenius norm. The reason for this is that we desire rotation-invariance: for details see [23].

4. Algorithmic aspects

We now move on to discuss the algorithmic aspects of the solution of the regularisation problems above. We do this through the saddle-point formulations.

Discretisation and the algorithm. We intend to apply the Chambolle-Pock algorithm [9] to the saddle-point form (S-TGV^2) of problem (P-TGV^2). This can be done after we discretise the original problem first; for the infinite-dimensional problem the (pre)conjugate K of K^* is not well-defined. We represent each tensor field f, u, w, φ and ψ with values on an uniform rectangular grid Ω_h of cell width $h > 0$, and discretise the operator div by forward differences with zero boundary conditions as div_h. We choose not to use central differences, because it tends to cause oscillation in this problem. This yields the discretised version

$$K_h^*(\varphi, \psi) := (-\operatorname{div}_h \varphi, -\varphi - \operatorname{div}_h \psi),$$

of the operator K^*. We then take $K_h := (K_h^*)^*$ as the discrete conjugate of K_h^*. It may be written

$$K_h(u, w) = (E_h u - w, E_h w)$$

for E_h the forward-differences discretisation of the operator E.

The algorithm may be stated as follows.

Perform the steps:

(1) Pick $\tau, \sigma > 0$ satisfying $\tau\sigma\|K_h\|^2 \leq 1$, as well as initial iterates (v^0, ξ^0). Set $\bar{v}^0 = v^0$.

(2) For $i = 0, 1, 2, \ldots$, repeat until a stopping criterion is satisfied.

$$\begin{aligned} \xi^{i+1} &:= (I + \sigma\partial F^*)^{-1}(\xi^i + \sigma K_h \bar{v}^i) \\ v^{i+1} &:= (I + \tau\partial G)^{-1}(v^i - \tau K_h^* \xi^{i+1}) \\ \bar{v}^{i+1} &:= v^{i+1} + \theta_i(v^{i+1} - v^i). \end{aligned}$$

The resolvent operators needed to calculate ξ^{i+1} and v^{i+1}, may be written

$$(I + \tau\partial G)^{-1}(v) = \arg\min_y \left\{ \frac{\|v - y\|^2}{2\tau} + G(y) \right\},$$

where for pairs $v = (u, w)$ we have to take $\|v\|^2 = \|u\|_F^2 + \|w\|_F^2$. The efficient realisation of Algorithm 4 depends on the efficient realisation of these minimisation problems. First, for $F^*(\varphi, \psi) = \delta_{\alpha V^3_{F,\mathrm{s}}}(\varphi) + \delta_{\beta V^4_{F,\mathrm{s}}}(\psi)$, the resolvent $(\varphi, \psi) = (I + \sigma\partial F^*)^{-1}(v, q)$ reduces to pointwise projection

$$\varphi(x) = P_{\alpha V^3_{F,\mathrm{s}}}(v(x)) \quad \text{and} \quad \psi(x) = P_{\beta V^4_{F,\mathrm{s}}}(q(x))$$

for all $x \in \Omega$, where

$$P_{\alpha V^k_{F,\mathrm{s}}}(z) := z \min\{1, \alpha/\|z\|_F\}.$$

This can be efficiently implemented on a parallel processor, such as a GPU.

Secondly, we have

$$G(u, w) = \frac{1}{2}\|f - u\|_{F,2}^2 + \delta_{\geq 0}(u),$$

for which we solve

$$[(I + \tau\partial G)^{-1}(u, w)](x) = \left(P_{\geq 0}\left(\frac{v(x) + f(x)\tau}{1 + \tau} \right), w \right), \quad (x \in \Omega).$$

The projection $x = P_{\geq 0}(z)$ to the positive semi-definite cone can be performed by projecting each eigenvalue γ_i, $(i = 1, \ldots, m)$ of z to $\mathbb{R}^+$, and reconstructing x with the projected eigenvalues and original eigenvectors. This follows from the optimality condition $z \in x + N_{\geq 0}(x)$, and the structure of the normal cone $N_{\geq 0}(x)$ at x to the positive semi-definite, described in [22, Lemma 3.1]. For $m = 2$ we may perform this operation by deriving explicit expressions, but for $m = 3$ such an appraoch is not numerically stable. We therefore need to employ the QR algorithm.

This is expensive, but can be completely per-voxel parallelised. Moreover, we use Sylvester's criterion to first check, whether we need to project.

In practise we initialise Algorithm 4 with $u^0 = 0$, $w^0 = 0$, $\varphi^0 = 0$, and $\psi^0 = 0$. We moreover approximate $\|K_h\|^2 \leq L^2 := (2\ell_m + 1 + \sqrt{1 + 4\ell_m})/2$, where $\ell_m = 4m/h^2$ (see [8]). In practise we choose $\sigma = \tau = 0.95/L$. As pointed out to us by a reviewer, speed-ups could potentially be obtained by choosing σ and τ unequal, or through the dimension-dependent steps of [16]. Our focus in this paper is, however, the advantages of a GPU over a CPU.

Thus, denoting by $F(\Omega_h; X) := \{g : \Omega_h \to X\}$ the space of functions from the discrete set Ω_h to X, we may expand Algorithm 4 for the present problem as follows.

Perform the steps:

(1) Initialise

$$u^0 = 0 \in F(\Omega_h; \mathrm{Sym}^2(\mathbb{R}^m)), \qquad w^0 = 0 \in F(\Omega_h; \mathrm{Sym}^3(\mathbb{R}^m)),$$
$$\varphi^0 = 0 \in F(\Omega_h; \mathrm{Sym}^3(\mathbb{R}^m)), \quad \text{and} \qquad \psi^0 = 0 \in F(\Omega_h; \mathrm{Sym}^4(\mathbb{R}^m)).$$

Set $\bar{u}^0 := x^0$ and $\bar{w}^0 := w^0$ and

$$\tau := \sigma := 0.95/\sqrt{(2\ell_m + 1 + \sqrt{1 + 4\ell_m})/2}, \quad \text{where } \ell_m = 4m/h^2.$$

(2) For $i = 0, 1, 2, \ldots$, repeat the following updates until a stopping criterion is satisfied.

$$\begin{aligned}
\varphi^{i+1}(x) &:= P_{\|\cdot\|_F \leq \alpha}\big(\varphi^i(x) + \sigma(E_h \bar{u}^i(x) - \bar{w}^i(x))\big)\\
\psi^{i+1}(x) &:= P_{\|\cdot\|_F \leq \beta}\big(\psi^i(x) + \sigma E_h \bar{w}^i(x)\big)\\
u^{i+1}(x) &:= P_{\geq 0}\left(\frac{u^i(x) + \tau \operatorname{div}_h \varphi^{i+1}(x) + \tau f(x)}{1 + \tau}\right)\\
w^{i+1}(x) &:= w^i(x) + \tau(\varphi^i(x) + \operatorname{div}_h \psi^i(x))\\
\bar{u}^{i+1} &:= u^{i+1} + \theta(u^{i+1} - u^i).\\
\bar{w}^{i+1} &:= w^{i+1} + \theta(w^{i+1} - w^i).
\end{aligned}$$

---

5. GPU implementation and performance

As mentioned in the introduction, instead of CUDA, we used OpenACC specification **#pragma** directives in normal C code to instruct a suitable compiler to generate GPU code. These directives are similar to those of OpenMP, and have the advantage over CUDA that in principle the same code can be compiled for OpenMP and single-CPU targets, reducing code maintenance effort. It should also enable more rapid development than manually writing CUDA GPU kernels. In the process of the GPU implementation of our code, we however ran into several restrictions of the current versions of the standard and the Portland Group PGCC compiler that was used, which we describe next.

The following code illustrates the usage of OpenACC.

```
#pragma omp parallel for
#pragma acc kernels loop deviceptr(y, f, res), independent
for(size_t i=0; i<n; i++)
    res[i]=y[i]+f[i];
```

OpenACC/PGCC difficulties: Code re-use. PGCC allows only selected function calls in kernel regions. In particular pointer parameters are often not allowed. This is a difficulty, because the C language only supports one-valued functions. We had to implement the function computing the two parameters $(c, s) = \text{givens}(a, b)$ of a Givens rotation as a macro, as we could not call the routine with prototype

```
void givens(double *c, double *s, double a, double b);
```

from the core function for the QR algorithm. A plausible reason for this restriction is that pointer-arithmetic should be possible on s and c, but the variables are stored in GPU registers, where this is not possible. Also, returning a **struct** containing c and s resulted in a failure. Partially this is thus a problem of the C language, although the compiler could try to deduce, what is really being done, and what language features are really needed.

Another difficult is that function pointers are not allowed, even though the compiler has the logic to inline functions passed as pointers, so that the pointer is actually never dereferenced in the machine code. The following example demonstrates such code-reuse, where a "paralleliser" multiop gets passed a primitive function as a pointer, to generate a parallelised version.

In the following code, without OpenMP or OpenACC enabled, both PGI PGCC and GNU GCC perfectly inline multiop below into multisqrt and multiexp, so that a pointer to sqrt and exp is never used in the machine code. With OpenMP both pass a function pointer to a non-inlined version of multip. PGCC refuses to generate OpenACC accelerator kernel.

```
void __inline__ multiop(double *p, size_t n, double (*f)(double)) {
    #pragma omp parallel for
    #pragma acc kernels loop deviceptr(p), independent
    for(size_t i=0; i<n; i++)
        p[i]=f(p[i]);
}

void multisqrt(double *p, size_t n) {
    multiop(p, n, sqrt);
}

void multiexp(double *p, size_t n) {
    multiop(p, n, exp);
}
```

This kind of restrictions severely limit code-reuse, although, clearly, the compiler possesses most of the optimization logic necessary to inline everything. Moreover, PGCC 12.5 has a bug in the implementation of the C99 **_Pragma** statement, so macros also cannot be used. (C **#pragma** may not be used in macros.) In the end, as the usual ways of code re-use resulted in failure, we had to manually inline code in separate files with **#include**, with parameters set with **#define**.

OpenACC difficulties: memory management in complex programs. It is our impression that the OpenACC GPU memory management model is designed only for programs with a simple nested structure, where it suffices to allow the compiler to copy data between GPU RAM and main RAM upon entry and exit of an accelerator region. However, in a more complex program, we cannot make the whole algorithm a single accelerator region, on the boundary of which the data is copied. Moreover, with large amount of data, the synchronisation upon frequent accelerator region entry and exit becomes inefficient, so we want a more detailed control over the storage and copying of data. OpenACC includes acc_malloc and acc_free to reserve and free memory in GPU RAM, but is missing an acc_memcpy procedure to efficiently copy memory when needed. This becomes useful, for one, when a task is split over multiple compute notes (with OpenMPI), and we want to copy the outermost data slices of a compute node to the neighbour node for differentiation of the outermost slices. During most of the execution of the algorithm, we do not need this data to reside in the main RAM, and never before the algorithm is finished, do we need the innermost slices in the main RAM. As OpenACC does not support such fine-grained memory management, we had to directly use cudaMemcpy from the CUDA API for this.

The following unabstracted program fits the OpenACC memory model.

```
void func(const char * infile ){
    size_t  sz= find_file_size ( infile );
    double *data=malloc(sz);
    read_doubles(data, sz,  infile );
    #pragma acc data copy(data[0:sz])
    // Calculate something
    free(data);
}
```

The following abstracted structure is not properly supported.

```
void func(const char * infile ){
    MyData *data=load_file( infile ); // Uses acc_malloc. (#pragma acc data
                                      // does not survive function return)
    // How to efficiently access the data on CPU (for, e.g., MPI sync)?
    // ==> Use cudaMemcpy!
    free_my_data(data);
}
```

OpenACC/PGCC difficulties: Manual optimizations. We need to efficiently store temporary 3×3 matrices for the QR code. It would be convenient to define these as C arrays, but that is inefficient on the GPU, as arrays cannot be stored in register. We therefore have to construct matrices out of individual **double** variables, and define macros to access these entries. The indices then have to be compile-time constants, so loops are not supported for operating on the matrix entries.

We would like to implement differentiation generically as

$$\begin{aligned} E_h u(x) &= A_{E,2}\mathrm{diff}(u,x), & \mathrm{div}_h\,\phi(x) &= A_{\mathrm{div},3}\mathrm{diff}(\phi,x),\\ E_h w(x) &= A_{E,3}\mathrm{diff}(w,x), & \mathrm{div}_h\,\psi(x) &= A_{\mathrm{div},4}\mathrm{diff}(\phi,x), \end{aligned}$$

for $\mathrm{diff}(\cdot,x)$ performing at x the forward-differences of every tensor component in every m directions, and each $A_{D,n}$ a suitable (sparse) prefix matrix. However, accessing these prefix matrices from GPU RAM is relatively slow, so we had to generate code with hard-coded constants for each supported differentiation operator. This results in relatively long compilation times to support multiple tensor orders and dimensions, and, for other regularisation functionals, both symmetric and non-symmetric gradient and divergence. Moreover, we also generate versions of the differentiation code with compile-time constant volume width and height for improved cache blocking for typical 128×128 and 256×256 dimensions. This results in further extended compile times.

Observe that differentation is still computationally expensive, since there are many tensor components per voxel: for $\mathrm{Sym}^4(\mathbb{R}^3)$, 16, for $\mathrm{Sym}^3(\mathbb{R}^3)$ 10, and for $\mathrm{Sym}^2(\mathbb{R}^3)$, 6 components. This implies for TGV^2 a total of $6 + 10 + 10 + 16 = 42$

Table 1. Performance of GPU and CPU hardware on 100 iterations of Algorithm 4 on a 128 × 128 ×60 data set. "CPU time" is the total execution time over all CPUs. "GPU exec time" denotes the time spent executing GPU code, while "GPU total time" includes memory access as well.

Hardware	Precision	Real time	CPU time	GPU exec time	GPU total time
2× 6-core Intel Xeon X5650 CPU	double	75.2s	898.5s		
	single	47.1s	562.8s		
1× Nvidia GeForce GTX 480 GPU	double	13.4s	13.4s	13.1s	13.3s
	single	5.2s	5.2s	5.0s	5.2s
1× Nvidia Tesla C2070 GPU	double	19.8s	19.7s	19.3s	19.8s
	single	7.8s	7.8s	7.4s	7.8s

Table 2. GPU performance advantage over 1×CPU core. The numbers on the left are speed-ups of the respective GPUs versus CPU. For the full run, relative decrease to 0.1% of the pseudo-duality gap was used as the stopping criterion. The reported CPU time for the full run is a projection estimate.

Hardware	double precision	single precision
1× GeForce	∼64×	∼108×
1× Tesla	∼45×	∼72×

Hardware	One iteration (double prec.)	Full run (1178 its.)
1× CPU core	9s	3h
1× Tesla	0.2s	3m50s

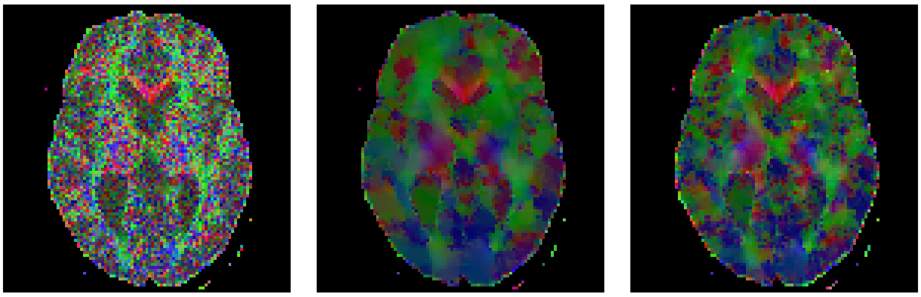
values per voxel, that need to be differentiated in all $m = 3$ directions at each iteration of Algorithm 4.

Performance. We computed 100 iterations of Algorithm 4 for a 128 × 128 ×60 data set. We need three instances of both the primal and dual variables, 3 × 42 doubles per voxel, yielding a 945MB memory requirement. This is all stored in GPU RAM for the GPU implementation. The raw run times are as reported in Table 1 The reported times include the computation of a pseudo-duality gap, not discussed here for the sake of conciseness; see [23] for details. A comparison of single CPU to single GPU performance is provided in Table 2. As we see, the performance improvement is noticeable. For double precision numbers, the GeForce GTX 480 GPU is about 64 times faster than a single CPU core, while the Tesla is about 45 times faster. This is well within the range of 10 to 100 times speed-up obtained in other applications with a manual CUDA implementation [11, 2, 18]. This indicates that good performance can be achieved with OpenACC. The improvement for single precision floating point numbers is even higher than for double precision. The overall algorithm however did not exhibit convergence using single precision numbers, suggesting accumulation of numerical errors.

For completeness, although analysis of the denoising results themselves is not the objective of this paper, we show in Figure 1 the result of denoising a very noisy text image by TV and TGV^2. Clearly TGV^2 performs better, with TV exhibiting staircasing: flat areas with sharp transitions. A more detailed comparison is presented in [23].

6. Conclusions

Our conclusion is that the application of GPUs offers significant speed-ups over CPUs. We would need 64 high-end CPU cores to match a single GPU. Regarding software tools, we find that the PGI implementation of OpenACC is still somewhat immature, although it does offer performance comparable to manual CUDA implementations. OpenACC should allow faster development of high-performance GPU



Noisy DTI image[1] TV denoising TGV^2 denoising

Figure 1. Results of TV and TGV^2 denoising.

code, and easier maintenance, as the same code can target OpenMP and unparallelised targets as well. Presently, however, much extra work is still needed, due to poor function support in kernels, and the OpenACC data region model, which does not appear suitable for programs with rich structure.

Acknowledgements

T. Valkonen has been financially supported by the SFB research program "Mathematical Optimization and Applications in Biomedical Sciences" of the Austrian Science Fund (FWF). The final publication is available at Springer via `http://dx.doi.org/10.1007/s00607-012-0277-x`.

The original diffusion-MRI measurement data on which Figure 1 is based on, is courtesy of Karl Koschutnig.

References

[1] Aksoy, M., Forman, C., Straka, M., Skare, S., Holdsworth, S., Hornegger, J., Bammer, R.: Real-time optical motion correction for diffusion tensor imaging. Magnetic Resonance in Medicine **66**(2), 366–378 (2011). DOI 10.1002/mrm.22787

[2] Amorim, R.M., Haase, G., Liebmann, M., dos Santos, R.W.: Comparing CUDA and OpenGL implementations for a Jacobi iteration. In: Smari and McIntire [18], pp. 22–32

[3] Arsigny, V., Fillard, P., Pennec, X., Ayache, N.: Fast and simple computations on tensors with log-euclidean metrics. Tech. Rep. 5584, INRIA (2005)

[4] Basser, P.J., Jones, D.K.: Diffusion-tensor MRI: theory, experimental design and data analysis – a technical review. NMR in Biomedicine **15**(7-8), 456–467 (2002). DOI 10.1002/nbm.783

[5] Bredies, K., Kunisch, K., Pock, T.: Total generalized variation. SIAM J. Imaging Sci. **3**, 492–526 (2011)

[6] Bredies, K., Kunisch, K., Valkonen, T.: Properties of L^1-TGV^2: The one-dimensional case. SFB-Report 2011-006, Karl-Franzens University of Graz (2011). URL `http://math.uni-graz.at/mobis/publications/SFB-Report-2011-006.pdf`

[7] Bredies, K., Valkonen, T.: Inverse problems with second-order total generalized variation constraints. In: Proceedings of SampTA 2011 – 9th International Conference on Sampling Theory and Applications, Singapore (2011)

[8] Chambolle, A.: An algorithm for total variation minimization and applications. J. Math. Imaging Vision **20**(1), 89–97 (2004)

[9] Chambolle, A., Pock, T.: A first-order primal-dual algorithm for convex problems with applications to imaging. J. Math. Imaging Vision pp. 1–26 (2010)

[10] Federer, H.: Geometric Measure Theory. Springer (1969)

[11] Freiberger, M., Egger, H., Liebmann, M., Scharfetter, H.: High-performance image reconstruction in fluorescence tomography on desktop computers and graphics hardware. Biomed. Opt. Express **2**(11), 3207–3222 (2011). DOI 10.1364/BOE.2.003207. URL `http://www.opticsinfobase.org/boe/abstract.cfm?URI=boe-2-11-3207`

[12] Gudbjartsson, H., Patz, S.: The Rician distribution of noisy MRI data. Magnetic Resonance in Medicine **34**(6), 910–914 (1995)

[13] Herbst, M., Maclaren, J., Weigel, M., Korvink, J., Hennig, J., Zaitsev, M.: Prospective motion correction with continuous gradient updates in diffusion weighted imaging. Magnetic Resonance in Medicine (2011). DOI 10.1002/mrm.23230

[14] Kingsley, P.: Introduction to diffusion tensor imaging mathematics: Parts I-III. Concepts in Magnetic Resonance Part A **28**(2), 101–179 (2006)

[15] Knoll, F., Bredies, K., Pock, T., Stollberger, R.: Second order total generalized variation (TGV) for MRI. Magnetic Resonance in Medicine **65**(2), 480–491 (2011). DOI 10.1002/mrm.22595

[16] Pock, T., Chambolle, A.: Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In: Computer Vision (ICCV), 2011 IEEE International Conference on, pp. 1762–1769 (2011). DOI 10.1109/ICCV.2011.6126441

[17] Setzer, S., Steidl, G., Popilka, B., Burgeth, B.: Variational methods for denoising matrix fields. In: J. Weickert, H. Hagen (eds.) Visualization and Processing of Tensor Fields, pp. 341–360. Springer (2009)

[18] Smari, W.W., McIntire, J.P. (eds.): 2009 International Conference on High Performance Computing & Simulation, HPCS 2009, Leipzig, Germany, June 21-24, 2009. IEEE (2009)

[19] Temam, R.: Mathematical problems in plasticity. Gauthier-Villars (1985)

[20] Tournier, J.D., Mori, S., Leemans, A.: Diffusion tensor imaging and beyond. Magnetic Resonance in Medicine **65**(6), 1532–1556 (2011). DOI 10.1002/mrm.22924

[21] Tschumperlé, D., Deriche, R.: Diffusion tensor regularization with constraints preservation. In: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), vol. 1, pp. 948–953. IEEE (2001)

[22] Valkonen, T.: Diff-convex combinations of Euclidean distances: a search for optima. No. 99 in Jyväskylä Studies in Computing. University of Jyväskylä (2008). Ph.D Thesis

[23] Valkonen, T., Knoll, F.: Total generalised variation in diffusion tensor imaging. SFB-Report 2012-003, Karl-Franzens University of Graz (2012). URL `http://math.uni-graz.at/mobis/publications/SFB-Report-2012-003.pdf`