J Real-Time Image Proc (2016) 11:127–139
DOI 10.1007/s11554-012-0312-3

ORIGINAL RESEARCH PAPER

# Performance engineering to achieve real-time high dynamic range imaging

**Harald Köstler · Markus Stürmer · Thomas Pohl**

Received: 27 April 2012 / Accepted: 8 December 2012 / Published online: 17 January 2013

**Abstract** Image-processing applications like high dynamic range imaging can be done efficiently in the gradient space. For it, the image has to be transformed to gradient space and back. While the forward transformation to gradient space is fast by using simple finite differences, the backward transformation requires the solution of a partial differential equation. Although one can use an efficient multigrid solver for the backward transformation, it shows that a straightforward implementation of the standard algorithm does not lead to satisfactory runtime results for real-time high dynamic range compression of larger 2D X-ray images even on GPUs. Therefore, we do a rigorous performance analysis and derive a performance model for our multigrid algorithm that guides us to an improved implementation, where we achieve an overall performance of more than 25 frames per second for 16.8 Megapixel images doing full high dynamic range compression including data transfers between CPU and GPU. Together with a simple OpenGL visualization it becomes possible to perform real-time parameter studies on medical data sets.

**Keywords** High dynamic range imaging · GPGPU · Multigrid · Performance model

H. Köstler (✉) · M. Stürmer
Universität Erlangen-Nürnberg, Lehrstuhl für Systemsimulation, Cauerstr. 11, 91058 Erlangen, Germany
e-mail: harald.koestler@informatik.uni-erlangen.de

M. Stürmer
e-mail: markus.stuermer@informatik.uni-erlangen.de

T. Pohl
Siemens AG, Healthcare Sector, Angiography and Interventional X-Ray Systems, Siemenstr. 1, 91301 Forchheim, Germany
e-mail: thomas.tp.pohl@siemens.com

## 1 Introduction

High dynamic range (HDR) imaging studies techniques that allow a wide dynamic range of luminances between the lightest and darkest areas of an image. In medical X-ray imaging, we face the problem that gray value intensities within objects like parts of the human body are much lower than outside where almost no X-rays are absorbed. Therefore, these bright areas outside the objects deteriorate the contrast within important areas for diagnostics. HDR compression is done by manipulation of the image gradients to improve the contrast of 2D X-ray images. However, since this is only one step within the image acquisition pipeline there are hard time constraints that have to be met in practical industrial applications. We address this on the one hand algorithmically by using an efficient multigrid solver for the arising linear systems and on the other hand by using GPUs that are currently one of the fastest hardware architectures for bandwidth-limited algorithms on structured data.

We demonstrate the structured development and engineering of a fast multigrid solver for imaging in the gradient domain on GPUs and its industrial application for HDR compression. The resulting software processes more than 25 frames of 16.8 MPixels per second on an Nvidia GTX 480 consumer GPGPU. As it is the main challenge to solve the Poisson equation as fast as possible, one could also think of applying our software to other problems like seamless image stitching [16] or painting [17].

The problem is that it is usually not sufficient to choose

1. a common HDR compression method [6],
2. an efficient multigrid solver [21], and
3. a fast and suitable hardware like a GPU,

but additionally it is necessary to adapt the multigrid algorithm and its implementation to the specific features of the hardware.

In order to obtain efficient software we will use a performance engineering approach that consists of

1. a rigorous analysis of the algorithm, possible implementation approaches, and target platforms,
2. development of a performance model for promising approaches,
3. the actual implementation, and eventually
4. measurement and analysis of the software performance and validation of the performance model.

Depending on the outcome of 4, it may be reasonable to adapt or refine the model, tweak the algorithm and its implementation, or—if the model(s) prove(s) to be based on wrong assumptions—start all over again.

This approach is a co-design of algorithm (components and parameters of multigrid method), its implementation (GPU kernels), and target platform (GPGPU) with respect to the application (gradient imaging).

In addition to giving better control of implementation process, this systematic approach including performance model, performance measurement and analysis enables other researches to estimate the quality of the implementation much better than raw numbers.

Therefore, one of the the main contributions of this paper is to outline the process of performance engineering and emphasize its benefits for real-time image processing applications. Furthermore, we introduce a new temporal shared memory blocking technique for our multigrid solver on GPU that fuses multiple steps of the algorithm.

In detail, we summarize in Sect. 2 the model for HDR compression in the gradient domain and show how to reconstruct the HDR compressed image from the gradient domain into the usual image domain by solving a simple partial differential equation (PDE). This is done numerically by a multigrid algorithm which is discussed in Sect. 3. For a proper performance engineering of the multigrid solver, we first select suitable multigrid components guided by local Fourier analysis and then outline and analyze three different implementation approaches in Sect. 4. Performance of the whole HDR compression algorithm and visual results on medical images are found in Sect. 5.

## 2 HDR compression in the gradient domain

We do HDR compression in the gradient domain as described in [6]. Here, the idea is to apply a position-dependent attenuating function $\Phi : \mathbb{R}^2 \mapsto \mathbb{R}$ to the gradient $\nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$ of an image $I : \Omega \mapsto \mathbb{R}$ defined in the rectangular image domain $\Omega \subset \mathbb{R}^2$. This results in compressed dynamic range image derivatives

$$C(x, y) = \nabla I(x, y)\Phi(x, y). \tag{1}$$

Note that we change only the magnitude of the gradients, but not their directions.

To increase the robustness of the method and to improve the visual quality of the results, the attenuating function $\Phi$ is computed on different image resolutions with the help of a Gaussian pyramid obtained by linear downscaling of the input image. The scaling factors $\phi_l(x, y)$ of the image gradients on each pyramid level $l = 0, \ldots, L$ are

$$\phi_l(x, y) = \frac{\alpha}{\|\nabla I_l\|}\left(\frac{\|\nabla I_l\|}{\alpha}\right)^{\beta} \tag{2}$$

where the first parameter $\alpha$ determines which gradient magnitudes are left unchanged, and the second parameter $\beta < 1$ is the attenuating factor of the larger gradients.

$\Phi$ is computed starting from the coarsest pyramid level $\Phi_L = \phi_L$ and then using linear interpolation $P$ recursively

$$\Phi_l = P(\Phi_{l+1})\phi_l \tag{3}$$

to obtain on the finest pyramid level, i.e., for the full image resolution, $\Phi = \Phi_0$.

In order to reconstruct the HDR compressed image from the changed image gradients, we are looking for an image $u : \Omega \mapsto \mathbb{R}$ minimizing the energy functional

$$\int_\Omega \|\nabla u - C\|^2 d\Omega. \tag{4}$$

A minimizer has to satisfy the Euler–Lagrange equation

$$\nabla^2 u = \mathrm{div} C. \tag{5}$$

Setting $f = \mathrm{div} C$, we thus have to solve the partial differential equation (PDE)

$$\Delta u = f \quad \text{in } \Omega \tag{6a}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{6b}$$

in order to compute $u$, where we assume Dirichlet boundary conditions.

For discretization of the image $I$ we use finite differences on a node-based grid $\Omega^h$ with mesh sampling size $h$, i.e., the distance between two neighboring grid points is $h$ in each direction. The discrete image reads $I^h : \Omega^h \mapsto G^h$. $G^h \subset \mathbb{R}$ denotes the gray value range which is typically [0..4095] for our application. The image derivatives are computed by simple forward and backward finite differences.

Equation (6) is also discretized by finite differences which leads to a linear system

$$A^h u^h = f^h , \quad \sum_{j \in \Omega^h} a^h_{ij} u^h_j = f^h_i, \quad i \in \Omega^h \tag{7}$$

with system matrix $A^h \in \mathbb{R}^{N \times N}$, unknown vector $u^h \in \mathbb{R}^N$

and right-hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete grid $\Omega^h$. $N$ denotes the total number of grid points and the number of unknowns in the linear system. In stencil notation the discretized Laplacian $\Delta^h = A^h$ reads on a uniform grid

$$A^h = \frac{1}{h^2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \tag{8}$$

Note that a stencil corresponds to one row $i$ of $A^h$, where the center point is the diagonal entry $a_{ii}$.

## 3 Multigrid

### 3.1 Multigrid basics

Equation (7) is solved numerically by a geometric multigrid method [1, 9]. As an iterative solver for large, sparse linear systems its major advantage is that it has an asymptotically optimal complexity of $\mathcal{O}(N)$, where $N$ is again the number of unknowns in the system [2, 21]. This theoretical efficiency can be combined with hardware-adapted and parallel algorithms to achieve the best possible performance like in [3, 8, 11, 14, 15, 19, 20]. In [13] we have shown that multigrid solvers can be used within variational approaches in medical image processing and computer vision problems like image denoising or optical flow.

If we define $u_h^*$ to be the exact solution of Eq. (7) and $u_h^{(k)}$ an approximation to it after $k$ iterations, the algebraic error is

$$e^h = u_h^* - u_h^{(k)}. \tag{9}$$

The multigrid idea is now based on two principles:

*Smoothing property* Classical iterative methods like Jacobi or Gauss–Seidel (GS) are able to smooth the error $e^h$ after very few steps. This means the high-frequency components of the error are removed well by these methods. But they have little effect on the low-frequency components. Therefore, the convergence rate of these methods is good in the first few steps and decreases considerably afterwards.

*Coarse grid principle* A smooth function on a fine grid can be approximated satisfactorily on a grid with less discretization points, whereas oscillating functions would disappear. Furthermore, a procedure on a coarse grid is less expensive than on a fine grid. The idea is now to approximate the low-frequency error components on a coarse grid.

Multigrid combines these two principles into a single iterative solver. One multigrid iteration, here the so-called *V-cycle*, is summarized in Algorithm 1. Starting on the finest level, first the *smoother*, typically a simple iterative method like GS, reduces the high-frequency error components in $\nu_1$ pre-smoothing iterations (step 4). Since usually the exact solution $u_h^*$ of the linear system is not known, we compute then the residual to estimate the quality of the current solution (step 5). Because of the linearity we have

$$A^h e^h = A^h\left(u_h^* - u_h^{(k)}\right) = f^h - A^h u_h^{(k)} = r^h. \tag{10}$$

The residual $r^h$ can now be restricted to the next coarser grid level with $H = 2h$ by averaging into $r^H$. The linear operator $R$ (step 6) is either the transpose of a linear interpolation (called full weighting $R_f$) taking into account all eight neighboring points in 2D or only the four direct neighbors excluding the diagonals (called half-weighting $R_h$).

After that this so-called residual equation

$$A^H e^H = r^H \tag{11}$$

is solved on the coarser grid level in order to approximate the low-frequency error components there (step 7). $A^h$ is represented by the stencil from Eq. (8) at each grid point, and on coarser levels we use rediscretization such that only the prefactor changes depending on the mesh sampling size.

The resulting error $e^H$ is then interpolated back to the finer grid by a linear interpolation operator $P$ (step 8) and eliminated there by the coarse grid correction (step 9). At the end $\nu_2$ post-smoothing iterations are performed (step 10). For more than two grid levels one obtains a recursive V-cycle which traverses between fine and coarse grids in a grid hierarchy.

**Algorithm 1** Recursive V-cycle: $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

```
1:  if coarsest level then
2:      solve $A^h u^h = f^h$ exactly or by many smoothing iterations
3:  else
4:      $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ // pre-smoothing
5:      $r^h = f^h - A^h \bar{u}_h^{(k)}$ // compute residual
6:      $r^H = R r^h$ // restrict residual
7:      $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ // recursion
8:      $e^h = P e^H$ // interpolate error
9:      $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ // coarse grid correction
10:     $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ // post-smoothing
11: end if
```

By nested iteration, the multigrid V-cycle algorithm can be extended to Full Multigrid (FMG). This means that we start with Eq. (6) on the coarsest grid level and compute a solution there using a low-resolution image obtained from the Gaussian image pyramid. This solution is then interpolated to the next finer level. Here, we can now do a V-cycle and use the interpolated solution as an initial guess. If we repeat these steps until we reach the finest grid level, ultimately only a small number of smoothing iterations independent of the number of unknowns must be performed on each level. Thus, multigrid can reach an asymptotically optimal complexity $\mathcal{O}(N)$ [21].

## 3.2 Choice of multigrid components

Since the optimal multigrid components depend on the operator $A^h$, there is a large number of possible smoothers and inter-grid transfer operators (restriction and prolongation) available, and even more ways to combine them into a working multigrid method. Choosing the right multigrid components is as important for the performance as their concrete implementation. It is impossible to evaluate all options, thus only a small number of alternatives is considered which have proven to yield good convergence as well as performance in literature and personal experience.

In a first step we derive asymptotic convergence rates by local Fourier analysis (LFA) for Eq. (6) to select the most promising multigrid components.

LFA is the main tool used for practical analysis of multigrid solvers and is based on (discrete) eigenfunctions or Fourier components that are for a constant coefficient infinite grid operator in 2D given by

$$\varphi^h(\theta,x) = \prod_{j=1}^{2} e^{\mathbf{i}\theta_j \frac{x_j}{h_j}}, \quad \theta \in \mathbb{R}^2, \quad \boldsymbol{x} \in \boldsymbol{\Omega^h} \tag{12}$$

with imaginary unit $\mathbf{i}$ and Fourier frequency $\theta$ on an infinite grid $\Omega^h = \{(\boldsymbol{x}) \mid x_i = z_i h_i, i \in \{1,2\}, z_i \in \mathbb{Z}\}$. The corresponding (discrete) eigenvalues or Fourier symbols are, e. g., for the discretized Laplacian

$$-\Delta^h(\theta) = \frac{1}{h^2}(4 - e^{\mathbf{i}\theta_1} - e^{-\mathbf{i}\theta_1} - e^{\mathbf{i}\theta_2} - e^{-\mathbf{i}\theta_2}) \tag{13}$$

$$= \frac{1}{h^2(4 - 2cos(\theta_1) - 2cos(\theta_2))}. \tag{14}$$

From this we are able to derive Fourier symbols for the smoother, the coarse grids, and the inter-grid transfer operators in order to compute an error reduction factor for a multigrid iteration. In order to evaluate, e.g., the symbol of the smoother $\hat{\mathcal{S}}$ in the whole domain, we note that Fourier components with

$$|\hat{\theta}| = \max\{|\hat{\theta}_1|, |\hat{\theta}_2|\} \geq \pi \tag{15}$$

are not visible on $\Omega^h$, since they coincide with components $\varphi_h(\theta,.)$, where $\theta = \hat{\theta}(mod 2\pi)$, due to the periodicity of the exponential function. Therefore, the Fourier space

$$\mathcal{F} = span\{\varphi_h(\theta,.) : \theta \in \Theta = ]-\pi,\pi]^2\} \tag{16}$$

contains any bounded infinite grid function $v^h \in \mathcal{F}(\Omega^h) = \{v^h \mid v^h(.) : \Omega^h \mapsto \mathbb{C}, \quad \|v^h\|_{\Omega^h} < \infty\}$. Since we use a grid hierarchy for multigrid, on coarser grids certain high frequencies coincide with low frequencies, depending on the coarsening scheme. We assume standard coarsening, i.e., the number of grid points is reduced by a factor of two in each dimension and decompose the Fourier space into low and high frequencies with $\Theta_{low} = ]-\pi/2,\pi/2]^2$ and $\Theta_{high} = \Theta \setminus \Theta_{low}$ as shown in Fig. 1. Then, a smoother acts mainly on high frequencies and the smoothing factor $\rho$ is defined as

$$\rho(\hat{\mathcal{S}}) = \sup_{\theta \in \Theta_{high}} \left\{ |\hat{\mathcal{S}}^h(\theta,\omega)| \right\}. \tag{17}$$

In Table 1 we list LFA results for a varying number of smoothing steps and two different restriction operators computed by the LFA software provided within [23]. Note that these predicted asymptotic error reduction factors are an upper limit for measured error reduction factors in our numerical experiments. The measured reduction factors depend on the initial solution and especially in the first few V-cycles the error reduction factors can be substantially better than the asymptotic error reduction factors.

Only V-cycles are taken into account, since although other cycle types like FMG are asymptotically better, they were less effective in our experiments. We will see that only a few multigrid iterations are required to reach a satisfactory solution within our application.

Furthermore, we choose the following components: as smoother $\mathcal{S}_h^\nu$, Gauss–Seidel with red–black ordering of the unknowns is used as shown in Fig. 2. In contrast to a usual lexicographic Gauss–Seidel, it leads to slightly better asymptotic convergence rates for our problem [23] and can be parallelized easily, because first the red unknowns can be updated in parallel and then the black ones.

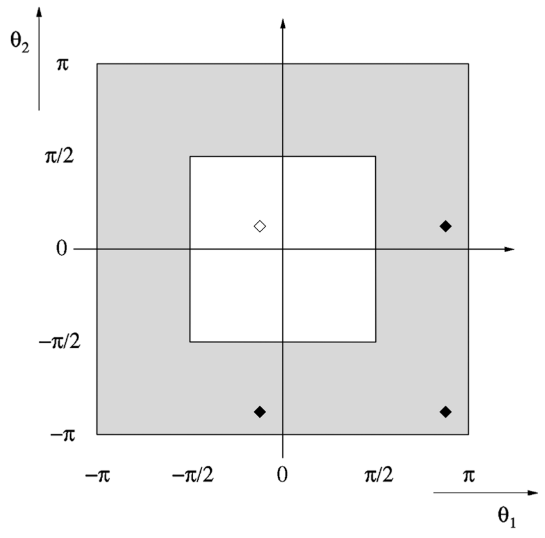

**Fig. 1** Fourier frequencies generating a space of 2h-harmonics, where $\Theta_{low}$ is the *white region* (the *white diamond* denotes a $\theta^{00} \in \Theta_{low}$) and $\Theta_{high} = \Theta \setminus \Theta_{low}$ is the *shaded region* (the *black diamonds* denote $\theta^{11}, \theta^{10}, \theta^{01} \in \Theta \setminus \Theta_{low}$) [23]

**Table 1** One-level smoothing factor $\rho$, spectral norm of two-level error reduction factor $\rho_{2L}$, and spectral norm of three-level error reduction factor $\rho_{3L}$ obtained by LFA for the 2D Poisson problem computed by the LFA software provided in [23]. We use a *red–black* Gauss–Seidel smoother, full- ($R_f$) or half-weighting ($R_h$) for restriction, and bilinear interpolation. $V(\nu_1, \nu_2)$ denotes a V-cycle with $\nu_1$ pre-smoothing and $\nu_2$ post-smoothing steps

| | $R$ | $V(1,1)$ | $V(2,1)$ | $V(2,2)$ | $V(3,2)$ |
|---|---|---|---|---|---|
| $\rho$ | | 0.063 | 0.033 | 0.025 | 0.019 |
| $\rho_{2L}$ | $R_f$ | 0.074 | 0.052 | 0.041 | 0.033 |
| $\rho_{2L}$ | $R_h$ | 0.104 | 0.074 | 0.056 | 0.044 |
| $\rho_{3L}$ | $R_f$ | 0.125 | 0.034 | 0.024 | 0.019 |
| $\rho_{3L}$ | $R_h$ | 0.161 | 0.053 | 0.034 | 0.027 |

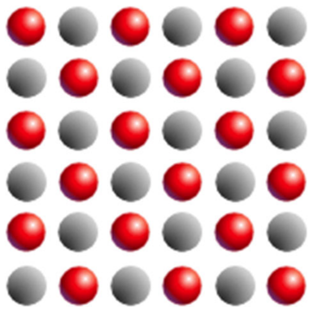

**Fig. 2** *Red–black* ordering of the grid points in 2D

LFA results, as shown in Table 1, suggest that half-weighting is preferable for more than two smoothing steps on each level and that more than five smoothing steps do not improve the convergence rate considerably. Therefore, we will use half-weighting and $V(2, 1)$, $V(2, 2)$, or $V(3,2)$ cycles in our further studies. For prolongation of the error, a bilinear interpolation operator $P$ is applied. For simplicity we restrict ourselves to squared domains of size $N = (2^L - 1)^2, L \in \mathbb{N}$ and thus have only one unknown on the coarsest grid such that we can easily solve there exactly.

Next we correlate for $k$ V-cycles the error reduction factor $\kappa = (\rho_\infty)^k$ (we choose $\rho_\infty = \rho_{3L}$ in the following) with the associated resource usage. Here, the resource of interest is time $T = k \cdot t_V$. This enables us to rate the efficiency of our multigrid solver and compare different multigrid settings, either according to the required resource usage

$$T = t_V \cdot log_{\rho_\infty} \kappa \tag{18}$$

to reach a common error reduction, or according to the error reduction factor

$$\kappa = (\rho_\infty)^{\frac{T}{t_V}} \tag{19}$$

which is achievable with a certain amount of resources.

In the next section we will estimate $t_V$ for concrete optimal GPU implementations of a multigrid V-cycle.

## 4 Performance engineering for the multigrid solver

This section follows the basic principles of performance engineering. It outlines and creates performance models first for a baseline implementation using a straightforward memory layout, then for an improved memory layout, and eventually for an advanced temporal blocking technique.

### 4.1 Preliminary considerations

It is known from previous work, e.g., [5] that most simple geometric multigrid methods are mainly memory bounded. Therefore, it is reasonable to aim for GPUs as target platforms, as they provide much higher bandwidth than general purpose CPUs at a better cost to bandwidth ratio. In the following, we focus on NVIDIA's Fermi architecture. The GPU code is developed using the Open Compute Language (OpenCL).

In contrast to its predecessors, Fermi relies on caches similar to most standard CPUs, and reading a single value requires to allocate a whole cache line. On the other hand, stencil operations on regular grids automatically exploit not only spatial, but even temporal locality, as multidimensional OpenCL work-groups typically map to rectangular regions.

For write operations Fermi can prevent allocation of partially written cache lines, but their eviction is about as expensive as for whole cache lines.

Hardware and algorithms are often characterized by the ratio between floating point operations and operand transfers. For all tests, an NVIDIA GTX 480 was used. It performs up to 1.35 GFlop/s with fused multiply-adds and has 177.4 GB/s peak memory bandwidth, leading to a balance of about 30.

The total runtime of a V-cycle

$$\begin{aligned} t_V &= \sum_{l=1}^{L} t_l = \sum_{l=1}^{L}\sum_{k=1}^{n_l} t_l^k = \sum_{l=1}^{L}\sum_{k=1}^{n_l} t(d_l^k) \\ &= \sum_{l=1}^{L}\sum_{k=1}^{n_l} \max\left(\frac{\mathrm{d}_\mathrm{l}^\mathrm{k}}{\mathrm{b}(\mathrm{d}_\mathrm{l}^\mathrm{k})}; \mathrm{t}_\mathrm{k}\right) \end{aligned} \tag{20}$$

comprises the times $t_l$ spent on each grid level $l$, which again consist of the execution times $t_l^k$ of the $n_l$ associated kernels. As main memory access dominates performance, $t_l^k$ will be modeled based on an estimate of how much data $d_l^k$ it transfers and with which bandwidth $b(d)$ this can optimally be done. Additionally, control and bookkeeping on the host system, taking place in the NVIDIA CUDA driver and the OpenCL runtime library, limits kernel throughput further. The observed runtime of a kernel is therefore at least $t_k$.

On each level, $A^h$ is given by the constant stencil from Eq. (8) and only the current approximation $u_h^*$ and the right-hand side $f^h$ of Eq. (7) need to be stored. For each of them, a storage size of $s_l$ is assumed, which will become the basis of the performance estimates. For simplicity, let also $v := v_1 + v_2$ be the total number of smoothing steps per level.

## 4.2 Considered optimizations

One iteration of the red–black Gauss-Seidel smoother consists of an update of the solution at all red grid points, followed by an update of the solution at all black grid points. In order to compute the new value of the solution at a certain grid point, the value of the right-hand side $f^h$ at that point, which is of the same color, and the value of the solution at the four direct neighbors, which are of the other color, are involved. It sets the local residual at the respective point to zero. This can be exploited in the following ways, as is also described in Algorithm 2.

Each resulting value of a half-weighting restriction depends on five residual elements. After at least one pre-smoothing step, four of them have just been set to zero. The restriction therefore degenerates to a weighted injection.

If at least one post-smoothing step is performed, it makes no sense to correct the red unknowns, as the succeeding red update will overwrite these results anyway. It is sufficient to correct only black unknowns.

A zero initial guess is used for coarse grid problems, and homogeneous Dirichlet boundary conditions are to be imposed. So the first red update on the coarse grid corresponds to a scaling of the red values of $f^h$ by the reciprocal of the diagonal matrix entry, which is constant on each grid level. This can be performed efficiently together with the construction of a coarse $f_H$.

If two or more red–black Gauss-Seidel iterations are used for pre-smoothing, one can even go further and omit initialization of the coarse-grid approximation and the first red update altogether. Instead, the first black update directly loads and scales the respective red values of $f^H$ as input. A second iteration of the smoother is necessary, as the red unknowns are otherwise still uninitialized.

It is assumed that the V-cycle descents to a coarse grid of 1 unknown, so that a single Gauss-Seidel iteration can be used as exact coarse grid solver. For these small grids, however, kernel call overhead becomes dominant over execution time. It, therefore, makes sense to handle all computations of the V-cycle below a certain level in a single kernel call employing a single OpenCL work-group. A reasonable threshold is at $31^2$ unknowns, because all required data will fit into the shared memory of the graphics card then (in OpenCL terminology this is local memory).

**Algorithm 2** Recursive V-cycle: $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

```
1: if coarsest level then
2:   solve $A^h u^h = f^h$ exactly for one grid point
3: else
4:   for $i = 1 \to \nu_1$ do
5:     if not finest level and $i = 1$ and $\nu_1 > 1$ then
6:       $\bar{u}_{h,b}^{(k)} = \mathcal{S}_h^b(diag(A^h)^{-1} f^{h,r}, A^h, f^{h,b})$ // pre-smoothing black with scaled input from $f^{h,r}$
7:     else
8:       $\bar{u}_{h,r}^{(k)} = \mathcal{S}_h^r(u_{h,b}^{(k)}, A^h, f^{h,r})$ // pre-smoothing red
9:       $\bar{u}_{h,b}^{(k)} = \mathcal{S}_h^b(\bar{u}_{h,r}^{(k)}, A^h, f^{h,b})$ // pre-smoothing black
10:    end if
11:  end for
12:  $f^H = R_h(f^{h,r} - A^h \bar{u}_{h,r}^{(k)})$ // compute and restrict residual
13:  $e^H = V_H(u^H, A^H, f^H, \nu_1, \nu_2)$ // recursion
14:  $\tilde{u}_{h,b}^{(k)} = \bar{u}_{h,b}^{(k)} + P e^H$ // interpolate error and do correct black unknowns
15:  for $i = 1 \to \nu_2$ do
16:    $u_{h,r}^{(k+1)} = \mathcal{S}_h^r(\tilde{u}_{h,b}^{(k)}, A^h, f^{h,r})$ // post-smoothing red
17:    $u_{h,b}^{(k+1)} = \mathcal{S}_h^b(u_{h,r}^{(k+1)}, A^h, f^{h,b})$ // post-smoothing black
18:  end for
19: end if
```

## 4.3 Basic memory layout

Each iteration of the *smoother* requires two separate calls to update the red and later the black unknowns due to the restricted synchronization capabilities of OpenCL's parallel paradigm. A basic memory layout will use one two-dimensional array for the current approximation $u_h^*$ and one for $f^h$. Even if only one in two values of each buffer is read or written, the hardware will have to effectively transfer each buffer completely. Each of the $2v$ kernel calls for the smoother will therefore transfer at least 3 $s_l$.

Except for the finest grid, the first red pre-smoothing update can be dropped, and the succeeding black update will use scaled red values from $f^H$ as input. These red values are brought into the cache with the black values, anyway, so that the first pre-smoothing requires only one kernel call transferring at least 2 $s_l$.

For half-weighting *restriction* the residuals need to be computed at all red points in only every second line, but half of the lines of $f^h$ and the whole $u_h^*$ are effectively read. Additionally, the coarse right-hand side $f_H$ needs to be written, leading to an estimate of 1.5 $s_l + s_{l+1}$.

*Correction* needs only to be applied to black fine grid unknowns, but effectively the whole fine grid $u_h^{(k)}$ is read and written. Together with the coarse grid approximation, this kernel transfers at least $2\ s_l + s_{l+1}$.

Looking only at transfer estimates, the simplification of the first pre-smoothing on coarse grids is the only algorithmic optimization this implementation draft profits from.

The execution times on each level $t_l$ from Eq. (20) become

$$t_1^{\text{basic}} = \overbrace{t(1.5s_1 + s_2)}^{\text{restriction}} + \overbrace{t(2s_1 + s_2)}^{\text{correction}} + \overbrace{2v \cdot t(3s_1)}^{\text{smoother}}$$
$$t_l^{\text{basic}} = \overbrace{t(1.5s_l + s_{l+1})}^{\text{restriction}} + \overbrace{t(2s_l + s_{l+1})}^{\text{correction}} + \overbrace{t(2s_l)}^{\text{first pre-smoothing}}$$
$$+ \underbrace{2(v-1) \cdot t(3s_l)}_{\text{rest of smoother}} \quad \text{for} \quad 1 < l < (L-4)$$
$$\sum_{l=L-4}^{L} t_l^{\text{basic}} = t_{31\times31}. \tag{21}$$

## 4.4 Improved memory layout with color splitting

As has become obvious in the previous section, the fact that red and black unknowns are stored intermixed results in unnecessary memory transfers. A common strategy is therefore to split the buffers for $u_h^{(k)}$ and $f^h$ each into two buffers that either hold red or black values. When referring to the respective buffers on a certain grid level, they are for simplicity denoted as $u_r$, $u_b$, $f_r$, and $f_b$. Each of those half-arrays requires $h_l \approx \frac{s_l}{2}$ of memory. The kernel structure and their tasks do not change, but the device memory transfer is reduced by this modification.

The *smoother* profits greatly from this memory layout. The update of one color of unknowns, for instance red, will only require reading $u_b$ and $f_r$ as well as writing $u_r$. This about halves the requirements of the basic memory layout to $3\ h_l \approx 1.5\ s_l$ in each kernel call. The first red update of pre-smoothing can still be dropped, but the succeeding black update will have the same transfer as any other—instead of the basic layout, the red values of $f^h$ must explicitly read from $f_r$.

For the *restriction*, red values for $u_h^*$ and $f_h$ are only required in every second line, but $u_b$ will need to be transferred completely, and both half-arrays of the coarse $f^H$ need to be written. Altogether, about $2\ h_l + 2\ h_{l+1} \approx s_l + s_{l+1}$ need to be transferred.

If constrained to black fine grid points, also *correction* profits from the split memory layout. Transfer is reduced to modifying black values of $u_h^*$ and reading the whole $u_H^{(k)}$, resulting in transfer of $2\ h_l + 2\ h_{l+1} \approx s_l + s_{l+1}$ just like for the restriction.

The time spent on each level now accumulates to

$$t_1^{\text{split}} = 2 \cdot \overbrace{t(s_1 + s_2)}^{\text{restriction/correction}} + \overbrace{2v \cdot t(1.5s_1)}^{\text{smoother}}$$
$$t_l^{\text{split}} = 2 \cdot \overbrace{t(s_l + s_{l+1})}^{\text{restriction/correction}}$$
$$+ \underbrace{(2v-1) \cdot t(1.5s_l)}_{\text{smoother}} \quad \text{for} \quad 1 < l < (L-4)$$
$$\sum_{l=L-4}^{L} t_l^{\text{split}} = t_{31\times31}. \tag{22}$$

## 4.5 Temporal shared memory blocking

Spatial cache blocking techniques, which adapt the processing order to improve cache locality, and temporal cache blocking techniques, that fuse multiple operations to increase locality further, have been proven to be efficient especially on standard CPUs and, to some extent, on GPUs [4, 7, 12, 18]. Especially for regular data structures, spatial cache blocking works equally simple in sequential and parallel codes and are implemented implicitly in OpenCL if multidimensional work-groups map to rectangular regions.

Utilizing temporal blocking techniques on GPUs is challenging. Coherency provided by OpenCL's memory model and synchronization facilites except inside work-groups are limited. On the other hand, a high degree of parallelism is essential. As a direct consequence of OpenCL's definition, work-groups must cooperatively compute the outcome of multiple operations and compose the result by writing disjoint regions. Using OpenCL's local memory (which will map to at most 48 kiB of shared memory per multi-processor in Fermi GPUs) is used as a common scratchpad for sharing intermediate results.

When fusing a whole red–black Gauss–Seidel update to $u_h^{(k+1)}$ into a single kernel call, a work group is responsible to compute new black values in a certain rectangular region $[\ x_1, x_2\ ] \times [\ y_1, y_2\ ]$. For simplicity it is assumed that this region is not close to the boundary. To be able to compute and store these black values, values of the preceding red update are required, in fact inside the larger region $[\ x_1 - 1, x_2 + 1\ ] \times [\ y_1 - 1, y_2 + 1]$, to satisfy data dependencies. These intermediate values in turn depend on the black values from $u_h^{(k)}$ within $[\ x_1 - 2; x_2 + 2\ ] \times [\ y_1 - 2; y_2 + 2\ ]$. Obviously, this halo of data dependencies and temporaries that need to computed redundantly by multiple work-groups grows further if fusing more operations.

To allow reasonable sizes, one-dimensional work-groups are used that compute only one line of such a region at a time. The respective procedure is depicted in Fig. 3 for

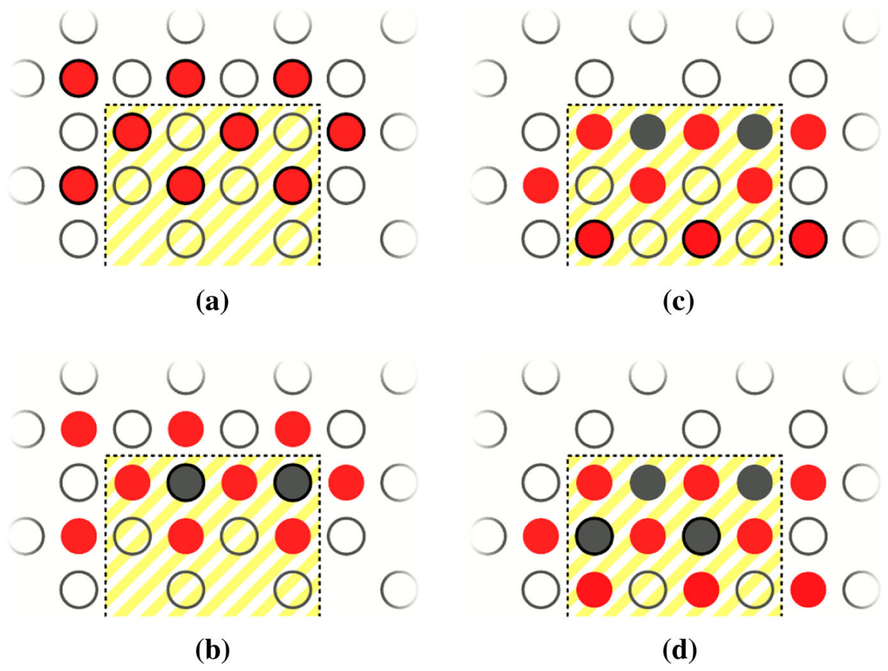

**Fig. 3** Temporal blocking is used to perform a *red–black* Gauss-Seidel iteration within the *shaded rectangular subdomain* of the grid. **a** Red unknowns from *three lines* need to be held in local memory as temporaries to compute black unknowns. **b** *One line of black values* can be computed within the rectangle and stored to global memory. **c** The oldest buffer of red temporaries is recycled for another partial line. **d** Another *line of black unknowns* can be updated

the temporally blocked smoother. Three partial lines of red intermediates need to be stored in local memory (see Fig. 3a) to enable the succeeding black update (see Fig. 3b). Like in a ring buffer, the oldest red line is then replaced (see Fig. 3c), allowing to compute the next black line (see Fig. 3d), and so on.

Fusing as many operations as possible and increasing the size of such regions to improve surface-volume-ratio can reduce transfer, but only until size of local memory and the required high degree of parallelism oppose. Eventually, it only makes sense until device memory bandwidth is not the limiting factor anymore. As intensive temporal blocking requires complicated control, introduces synchronization and heavily accesses shared memory and cache, this point can be reached very soon.

In the smoother, red points only show up as intermediate values—in fact, this section will outline an implementation that uses a split memory layout that does not store the zero-valued boundary conditions and handles all red unknowns on all levels as temporaries, so that red values need only to be written in the very last post-smoothing step of the last V-cycle to create a complete solution.

By fusing red and black update, *smoothing* only requires $\nu$ kernel calls. Each will need to read the stored black half of $u_h^{(k)}$ and the whole $f^h$ to write a new black half of $u_h^*$, so that the estimate becomes $4\ h_l \approx 2\ s_l$. The initial smoothing step on coarser grid levels only depends on $f^H$; hence transfer is reduced to $3\ h_l \approx 1.5\ s_l$.

The *restriction* is always performed together with the last pre-smoothing step. In addition to the three red lines required for the smoothing part, three lines of black need to be preserved for the residual calculation and the dependency halo is increased by another layer. On the other hand, no additional kernel call is added. As only the coarse $f_H$ needs to be written in addition to pure smoothing, the combined kernel transfers accumulates to about $4\ h_l + 2\ h_{l+1} \approx 2\ s_l + s_{l+1}$. If only a single pre-smoothing step is used—such V-cycles are not examined—this is reduced by another $0.5\ s_l$.

Similarly, the correction of level $l - 1$ is combined with the last post-smoothing step on level $l$. As the coarse iteration $u_h^*$ is not required further, its values are used for correction and then discarded. Concerning the coarse grid, only the black values of $u_H^{(k)}$ and the whole $f_H$ are read, and the transfer sums up to $3\ h_l + 2\ h_{l\text{-}1} \approx 1.5\ s_l + s_{l\text{-}1}$.

Please note that here correction contributes to the runtime of the coarser level. As the red intermediates are required for fine grid correction, the specialized kernel for descending below $31 \times 31$ will skip the last post-smoothing. Instead, a kernel call performing that Gauss-Seidel update and correcting the $63 \times 63$ level is added.

The time spent on each level is now estimated as

$$t_1^{\text{block}} = \overbrace{t(2s_1 + s_2)}^{\text{last pre-smoothing with restriction}} + \overbrace{(\nu - 1)\cdot t(2s_1)}^{\text{rest of smoother}}$$

$$t_l^{\text{block}} = \overbrace{t(1.5s_l)}^{\text{first pre-smoothing}} + \overbrace{t(2s_l + s_{l+1})}^{\text{last pre-smoothing with restriction}}$$
$$+ \overbrace{t(1.5s_l + s_{l-1})}^{\text{last post-smoothing with correction}}$$
$$+ \underbrace{(\nu - 3)\cdot t(2s_l)}_{\text{rest of smoother}} \quad \text{for } 1 < l < (L-4)$$
$$\sum_{l=L-4}^{L} t_l^{\text{block}} = t_{31\times 31} + \underbrace{t(1.5s_{L-4} + s_{L-5})}_{\text{correction of level}(L-5)}. \tag{23}$$

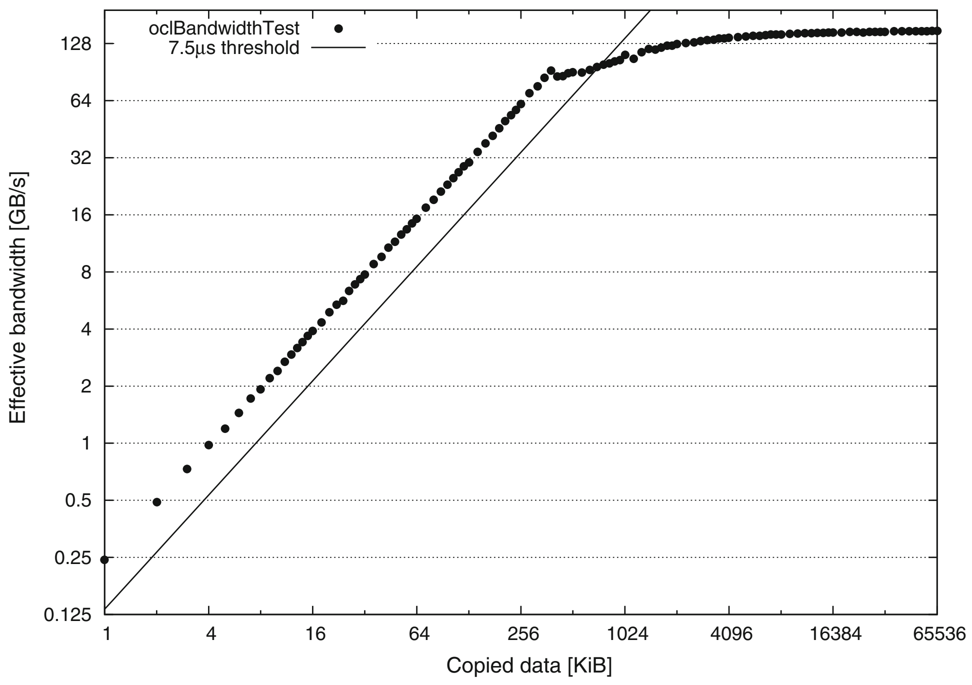

**Fig. 4** Observed bandwidth depending on size of copied data and kernel throughput on Nvidia GTX 480

## 4.6 Evaluation of implementation approaches

As gold standard for memory bandwidth, the device-to-device copy performance of the *clEnqueueCopyBuffer* command is used. Results for the GTX 480 are depicted in Fig. 4. For small sizes, the observed bandwidth only depends on the maximum throughput of copy operations, which is roughly one per 4 $\mu s$ on the test system. Starting from about 256 KiB, the actual operation on the card becomes dominant, reaching a peak bandwidth of slightly below 150 GB/s at 64 MiB. Consider that data needs to be read and written for copying, so actually twice as much data was transferred.

The possible kernel throughput varies with, among other factors, number and type of arguments, and can be derived experimentally. As a reference, executing an empty OpenCL kernel taking four buffer and four integral arguments using a single work-item was chosen, resulting in a throughput of about one kernel call every 7.5 $\mu s$. For small data sizes, this imposes a stricter limit than the built-in copy performance, the corresponding bandwidth limitation is also included in the graph.

The specialized 31 $\times$ 31 coarse grid kernel cannot be reasonably modeled by its device memory throughput. As only a single work group is started, only a small part of the GPGPU is active—a single of 14 so-called streaming multiprocessor. Memory operations that are cached or to local memory become similar slow as global memory accesses, frequent synchronization limits performance further. Thus we chose to use the measured runtime of our implementation for a coarse $V(2,1)$ kernel call as $t_{31\times 31}$, which is 30 $\mu s$.

An intuitive understanding is possible when looking at the performance estimates for typical setups. Eventually, one is interested in reducing the error as fast as possible. Based on Eq. (18), $t_{1E-4}$ estimates the time to reduce the error by four magnitudes, which requires 3.14 $V(2,1)$-cycles, 2.72 $V(2,2)$-cycles, or 2.55 $V(3,2)$-cycles. This comparison is equally valid for any other target accuracy due to the monotonicity of the logarithm.

For problems of $1{,}023^2$, $2{,}047^2$, and $4{,}095^2$, Table 2 compares the runtime and time-to-solution efficiency that can be expected for three V-cycle configurations.

Here, the $V(2,1)$-cycle promises fastest convergence for all outlined implementations. But as the performance of a final implementation will vary to some extent, this will have to be re-evaluated with measured performance values again.

## 4.7 Performance results for multigrid solver with temporal blocking

Eventually, temporal blocking with a split memory layout was implemented in OpenCL as outlined before. Performance measurements are included in Table 2. Also for the real execution time, the $V(2,1)$ proves to be optimal.

The actual runtime is about twice as long as the estimate at grid size $1{,}023^2$, about 1.6 times as long at $2{,}047^2$, and about 1.4 times as long at $4{,}095^2$. There are various reasons and explanations for these deviations:

**Table 1** Performance estimates and measurements for implementation of temporal blocking in ms

| | $t_V$ | | | $t_{1E-4}$ | | |
|---|---|---|---|---|---|---|
| | $V(2,1)$ | $V(2,2)$ | $V(3,2)$ | $V(2,1)$ | $V(2,2)$ | $V(3,2)$ |
| $1,023^2$ | | | | | | |
| Basic layout | 1.03 | 1.31 | 1.59 | 3.2 | 3.6 | 4.1 |
| Split layout | 0.68 | 0.85 | 1.03 | 2.1 | 2.3 | 2.6 |
| Blocking | 0.43 | 0.54 | 0.64 | 1.4 | 1.5 | 1.6 |
| Measurement | 0.90 | 1.10 | 1.30 | 2.8 | 3.0 | 3.3 |
| $2,047^2$ | | | | | | |
| Basic layout | 3.4 | 4.4 | 5.4 | 10.7 | 12.0 | 13.7 |
| Split layout | 2.0 | 2.5 | 3.0 | 6.1 | 6.7 | 7.6 |
| Blocking | 1.3 | 1.6 | 2.0 | 4.1 | 4.4 | 5.0 |
| Measurement | 2.1 | 2.6 | 3.2 | 6.6 | 7.2 | 8.1 |
| $4,095^2$ | | | | | | |
| Basic layout | 12.9 | 16.5 | 20.2 | 40.3 | 45.0 | 51.5 |
| Split layout | 7.0 | 8.8 | 10.7 | 21.9 | 24.1 | 27.3 |
| Blocking | 4.7 | 5.9 | 7.1 | 14.6 | 16.1 | 18.2 |
| Measurement | 6.5 | 8.3 | 10.0 | 20.5 | 22.5 | 25.4 |

The performance models are optimistic in the assumption of perfect caching, which is not feasible in practice. Further, an algorithm that requires complex addressing and relies on data reuse is likely not to saturate the memory bus as good as a simple copy operation. Both are especially true for temporal blocking, where a resulting value depends on a larger neighborhood and where the ratio between device memory access and work inside the multiprocessor gets even worse.

Using the NVIDIA visual profiler (nvvp), the behavior of the $V(2,1)$-cycle for $2047^2$ is now investigated in more detail. An extensive description of the Fermi architecture can be found in OpenCL Programming Guide, 2012, appendix C; an exact description of the events and metrics supported by nvvp can be found in CUPTI User's Guide (2012).

Table 3 shows the measured bandwidth and the excessive transfer, i.e., how much more memory transfer was measured than estimated in the optimistic performance model, for the smoothing kernel at $2,047^2$, for the kernel call that performs smoothing with successive restriction to $1,023^2$, and for the smoothing at $1,023^2$ with correction of the finer grid of $2,047^2$. The two questions remaining are if the excessive transfer is reasonable and why the achieved memory bandwidth is below prediction.

The excessive transfer depends on the block size used by the blocking technique, which was determined experimentally to yield best performance. To get a simple yet reasonable estimate the effects at the boundaries and of caching between work-groups are neglected, and optimal caching inside work-groups is assumed. A single work-group performing the *smoothing* as analyzed in Table 3 will read $193 \times 26$ elements of $u_b$ and $192 \times 24$ of $f_r$ for the red update from device memory first, and then $191 \times 22$ elements of $f_b$ to compute and write as many new black iterates. Considering that always whole cache lines comprising 32 values are loaded and the transfer estimate of $4 \times 191 \times 22$ values from the model, an excess of

**Table 3** Analysis of runtime, bandwidth, and amount of excessive memory transfer for smoothing at grid size $2,047^2$ alone and with restriction to $1,023^2$, and for smoothing at $1,023^2$ with successive correction

| | Runtime (ms) | Bandwidth (GB/s) | Excess (%) |
|---|---|---|---|
| Smoothing | 0.31 | 124 | 13 |
| Smoothing with restriction | 0.42 | 108 | 19 |
| Smoothing with correction | 0.26 | 112 | 14 |

$$\frac{\overbrace{224\cdot 26}^{\text{read } u_b}+\overbrace{192\cdot 24}^{\text{read } f_r}+\overbrace{192\cdot 22}^{\text{read } f_b}+\overbrace{191\cdot 22}^{\text{write } u_b}}{\underbrace{4(191\cdot 22)}_{\text{model estimate}}}-100\% \approx 12.2\%$$

should be expected, and using the same approach 14.5 % excess for *smoothing with restriction* and 13.0 % for *smoothing with correction*. This also proves that the shared memory blocking method actually works.

NVIDIA's Fermi architecture provides a dual-issue pipeline, resulting in a peak performance for each processing element of two instructions per cycle (IPC). Additionally, operations that cannot complete in time may need to be replayed—e.g., because of memory bank collisions when accessing cache or shared memory or cache misses.

The profiler measures an IPC value of roughly 1.25 for *smoother* and about 1.45 for *smoother with restriction*. An instruction replay of close to 20 % in the first case and more than 10% in the latter indicate an even better utilization. Also other metrics, e.g., low thread divergence (below 4 %) and high achieved occupancy (over 95 %), imply that the kernels execute efficiently on large grids.

Reducing the number of instructions further is difficult. One can retrieve the intermediate code that is generated from the OpenCL runtime and passed to the graphics driver, but the final binary code can neither be extracted nor are its instructions publicly documented. Experience, guess-work, and trial-and-error have already been largely invested to optimize the kernels.

Looking at small grids, however, the picture changes. For $511^2$ unknowns, the smoother achieves only below 16 GB/s of device memory bandwidth instead of the model

**Table 4** Achievable frames per second (fps) for HDR compression with different image sizes and runtime in ms for transferring the image to GPU and back, doing the HDR compression, and restoring the resulting image

| $N$ | fps | Time transfer | Time HDR | Time solver |
|---|---|---|---|---|
| $1{,}023^2$ | 240 | 0.73 | 0.71 | 2.70 |
| $2{,}047^2$ | 88 | 2.78 | 2.22 | 6.32 |
| $4{,}095^2$ | 26 | 10.8 | 8.01 | 19.6 |

value of 110 GB/s. As there are not even enough work groups to employ all multiprocessors, the occupancy drops below 15 %.

It was shown that the temporal blocking technique is able to reduce the memory transfer drastically and reach great performance on large grids, even better than a straightforward implementation can optimally achieve. On the other hand, it requires substantial control code on GPGPUs making instruction throughput a limiting factor.

For large grids the model delivers reasonable estimates, with explainable deviations. As the optimized solver will be used interactively, performance is critical for large images, and sub-optimal performance for small images still produces sufficient frame rates. Otherwise, we had to adapt the performance model to better catch the effects on smaller grid levels and probably switch to another implementation there.

## 5 HDR compression results

In order to do full HDR compression, we have to

1. transfer the input image from CPU to GPU via the PCIe bus,
2. apply the attenuating function $\Phi$ as described in section 2 to its derivatives,
3. restore the HDR compressed image by solving Eq. (6) via three V(2,1)-multigrid cycles, and
4. transfer the final image back from GPU to CPU via the PCIe bus.

In our application we have 2D X-ray images with a gray value range of 12 bit, i.e., [0..4095], stored as short values. In order to guarantee subpixel accuracy for the backward transform for such images, i.e., the maximum pixel error between the original and the restored image is below 1, we have to perform three $V(2,1)$-cycles. Note that the reduction of the maximal pixel error roughly is in the same order of magnitude as the reduction of the residual. We fit the images of arbitrary sizes to the next higher power of two numbers of pixels in each dimension in order to use our efficient multigrid solver. For the transfers and the HDR compression the actual image sizes can be used.

The transfer time can be easily approximated with the measured bandwidth of the PCIe bus that is for pinned memory approximately 6 GB/s and the fact that for each pixel $2 \cdot 2 = 4$ Bytes have to be transferred to GPU and back.

The HDR compression part basically computes the right-hand side for the solver and is implemented within two GPU kernels. The main kernel loads the original image on each level from global memory and writes the attenuating function $\Phi$ and the RHS. This kernel is very compute intensive since for each pixel besides approximately 20 add or multiply operations and two divisions also a square root and power have to be computed, where we apply the *native* variants of these functions.

In addition to the main kernel we require a bilinear interpolation kernel for $\Phi$ (see Eq. 3). Note that we did not further optimize the HDR compression part since it shows to contribute only a small fraction to the overall runtime.

Table 4 lists the overall performance of HDR compression for different image sizes and splits the runtime into the fractions for CPU-GPU transfers, HDR compression and solving. For the largest size $4{,}095^2$ that shows best performance roughly 51 % of the time is spent in the solver, 28 % in the transfers, and 21 % in the HDR compression. Next we show test results for two HDR compressed 2D X-ray images in Fig. 5.

The HDR compression is part of a software package that supports different image formats and provides additional functionality like an interactive OpenGL visualization of the HDR compressed images on GPU. This enables the user to do parameter studies for a series of input images. Additionally, post processing like windowing of the gray values can be done.

## 6 Discussion

Of course, besides multigrid there exist other methods to solve the Poisson equation like FFT based techniques or conjugate gradients, but it is known that they have asymptotically worse convergence rates. Since the computational effort for one grid point within a multigrid solver is constant independently of the problem size $N$, i.e., the computational complexity is $\mathcal{O}(N)$, at least for larger problems multigrid will be faster than FFT-based solvers having computational complexity $\mathcal{O}(N \log N)$. In [22] a discrete sine transform (DST)-based Poisson solver for HDR compression is developed and in [19] we compared multigrid and FFT for the 3D Poisson equation for imaging problems and estimated the problem size, for which multigrid becomes faster than FFT.

Besides these theoretical considerations it is difficult to compare different implementations of solvers. Of course, we could provide runtimes for other solvers, but it would

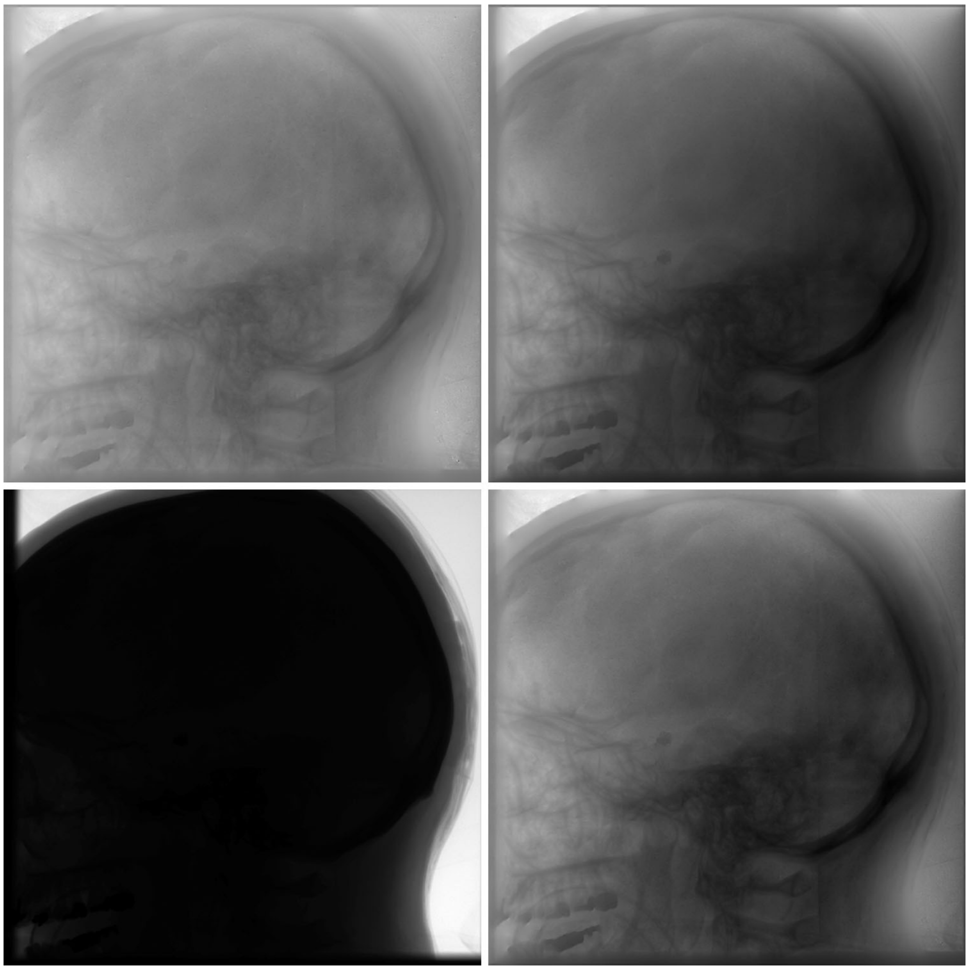
**Fig. 5** HDR Compression results for a 2D X-ray image of size 960 × 960. Data provided by Prof. Dr. R. Chapot (Klinik für Neuroradiologie und endovaskuläre Therapie, Alfred Krupp Krankenhaus, 45131 Essen). The original images (*lower left*) are restored using three $V(2,1)$-cycles and the parameters for HDR compression were $\alpha = 0.1 \cdot$ *average image gradient* and $\beta = 0.45$ (*upper left*) $\beta = 0.55$ (*lower right*), and $\beta = 0.65$ (*upper right*)

only be a fair comparison if these codes were also optimized towards our problem and the used hardware. To roughly give an idea of the performance of an FFT-based multigrid solver for our problem, we run a forward and backward FFT using the cuFFT[1] library that is one of the fastest FFT implementations on GPU currently. In order to solve the Poisson equation with Dirichlet boundary conditions a DST is necessary [22] that can be computed via an FFT of size $2N$. We neglect the time for dividing the coefficients before the backward transformation. For a problem size $1{,}023^2$ cuFFT takes 2.22 ms, for $2{,}047^2$ it takes 8.61 ms, and thus the multigrid solver is already faster since it takes 6.32 ms in that case (see Table 4).

## 7 Conclusions

We have implemented a tool to do real-time imaging in the gradient domain based on an efficient multigrid solver on GPU. As an example, we show that on current GPUs real-time HDR compression of medical data sets up to size $4{,}095^2$ is possible. In order to achieve an optimal implementation we apply a structured performance engineering approach based on a detailed performance model.

Next we plan to compare our performance on NVIDIA GPUs to AMD graphics cards and CPU implementations of OpenCL. Furthermore, the algorithm can be extended easily to 3D medical data sets, where applicability and efficiency of temporal blocking techniques is interesting.

Additionally, alternative HDR compression methods based on filters like in [10] can be compared with the implemented method.

[1] http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf.

## References

1. Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comput. **31**(138), 333–390 (1977)
2. Briggs, W., Henson, V., McCormick, S.: A Multigrid Tutorial, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA (2000)
3. Christadler, I., Köstler, H., Rüde, U.: Robust and efficient multigrid techniques for the optical flow problem using different regularizers. In: Hülsemann F, Kowarschik M, Rüde U (eds)

Proceedings of 18th Symposium Simulationstechnique ASIM 2005, SCS Publishing House, Erlangen, Germany, Frontiers in Simulation, vol. 15, pp. 341–346, preprint version published as Tech. Rep. 05-6 (2005)
4. Christen, M., Schenk, O., Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: In International Parallel and Distributed Processing Symposium (IPDPS), IEEE, pp. 676–687 (2011)
5. Douglas C., Hu J., Kowarschik M., Rüde U., Wei C.: Cache optimization for structured and unstructured grid multigrid. Electr. Transact. Numer. Anal. (ETNA) 10:21–40 (2000)
6. Fattal R., Lischinski D., Werman M.: (2002) Gradient domain high dynamic range compression. ACM Transact. Graph. 21(3): 249–256
7. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Arvind, Rudolph L (eds.) Proceedings of the 19th annual international conference on Supercomputing, ACM, pp 361–366 (2005)
8. Gradl, T., Freundl, C., Köstler, H., Rüde, U.: Scalable multigrid. In: Wagner, S., Steinmetz, M., Bode, A., Brehm, M. (eds) High Performance Computing in Science and Engineering. Garching/ Munich 2007, LRZ, KONWIHR, Springer, Berlin, pp. 475–483 (2008)
9. Hackbusch, W.: Multi-Grid Methods and Applications. Springer, Berlin (1985)
10. He, K., Sun, J., Tang, X.: Guided image filtering. Computer Vision–ECCV 2010 pp. 1–14 (2010)
11. Hülsemann, F., Kowarschik, M., Mohr, M., Rüde, U.: Parallel geometric multigrid. In: Bruaset A, Tveito A (eds) Numerical Solution of Partial Differential Equations on Parallel Computers, Lecture Notes in Computational Science and Engineering, vol. 51, Springer, Berlin, chap. 5, pp. 165–208 (2005)
12. Kamil, S., Chan, C., Oliker, L., Shalf, J., Williams, S.: An autotuning framework for parallel multicore stencil computations. In: In International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–12 (2010)
13. Köstler, H.: A Multigrid Framework for Variational Approaches in Medical Image Processing and Computer Vision. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (2008)
14. Köstler H., Stürmer M., Rüde U.: A fast full multigrid solver for applications in image processing. Numerical linear algebra with applications **15**(2–3):187–200 (2008)
15. Kowarschik, M., Wei, C., Rüde, U.: DiMEPACK: a cache–optimized multigrid library. In: Arabnia H (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), CSREA Press, Irvine, CA, USA, Las Vegas, NV, USA, vol. I, pp. 425–430 (2001)
16. Levin, A., Zomet, A., Peleg, S., Weiss, Y.: Seamless image stitching in the gradient domain. Lecture Notes in Computer Science, Computer Vision-ECCV 2004 3024/2004:377–389 (2004)
17. McCann, J., Pollard, N.: Real-time gradient-domain painting. ACM Transact. Graphics-TOG **27**(3):93–93 (2008)
18. Schäfer, A., Fey, D.: High Performance Stencil Code Algorithms for GPGPUs. Procedia Comput. Sci. **4**, 2027–2036 (2011)
19. Stürmer, M., Treibig, J., Rüde, U.: Optimising a 3d multigrid algorithm for the IA-64 architecture. Int. J. Comput. Sci. Eng. **4**(1), 29–35 (2008)
20. Stürmer, M., Wellein, G., Hager, G., Köstler, H., Rüde, U.: Challenges and potentials of emerging multicore architectures. In: Wagner, S., Steinmetz, M., Bode, A., Brehm, M. (eds.) High Performance Computing in Science and Engineering. Garching/ Munich 2007, LRZ, KONWIHR, Springer, Berlin, pp. 551–566 (2008)
21. Trottenberg, U., Oosterlee, C., Schüller, A.: Multigrid. Academic Press, San Diego, CA, USA (2001)
22. Wang, T., Ke, W., Zwao, D., Chen, F., Chiu, C: Block-based gradient domain high dynamic range compression design for real-time applications. In: Image Processing, 2007. ICIP 2007. IEEE International Conference on, IEEE Computer Society Washington, DC, USA, vol. 3, pp. 561–564 (2007)
23. Wienands, R., Joppich, W.: Practical Fourier analysis for multigrid methods, Numerical Insights, vol. 5, Chapmann and Hall/ CRC Press, Boca Raton, Florida, USA (2005)

## Author Biographies

**Harald Köstler** finished his Ph.D. in 2008 on multigrid methods in medical image processing. Currently, he is a postdoctoral researcher at the Chair for System Simulation at University of Erlangen-Nuremberg. His research interests include performance and software engineering concepts for simulation software and real-time image processing, multigrid methods and programming techniques for parallel hardware, especially GPUs and heterogeneous GPU-CPU clusters.

**Markus Stürmer** Dipl.-Inf. Markus Stürmer is research assistant in the system simulation group of Prof. Dr. Ulrich Rüde at University of Erlangen-Nuremberg and working on his doctoral thesis about performance engineering for multi- and manycore systems.

**Thomas Pohl** has a degree in physics from the University of Augsburg where he worked on computer simulations for solid state physics. He also holds a PhD in computer science from the University of Erlangen-Nuremberg in the field of HPC and computational fluid dynamics. Currently, he is working for Siemens Healthcare as an innovation manager and technical lead to develop a new prototyping platform for medical applications in the context of interventional procedures.