# Efficient Finite Element Geometric Multigrid Solvers for Unstructured Grids on GPUs

Markus Geveler, Dirk Ribbrock, Dominik Göddeke, Peter Zajac, and Stefan Turek

Applied Mathematics (LS3), TU Dortmund, Germany,
markus.geveler@math.tu-dortmund.de

**Abstract.** Fast, robust and efficient multigrid solvers are a key numerical tool in the solution of partial differential equations discretised with finite elements. The vast majority of practical simulation scenarios requires that the underlying grid is unstructured, and that high-order discretisations are used. On the other hand, hardware is quickly evolving towards parallelism and heterogeneity, even within a single workstation. Commodity CPUs have multiple cores, and GPUs are the most prominent example of current fine-grained parallel architectures.

We are convinced that geometric multigrid methods are superior to algebraic multigrid methods, *if* their components are designed with respect to the underlying finite element discretisation. Such an approach, which we call *finite element geometric multigrid (FE-GMG)*, allows the design and development of numerically optimal solvers. While many multigrid components can be parallelised in a straight forward manner, two components pose severe challenges: Robust and strong smoothers are inherently recursive and sequential, and grid transfer operations (prolongation and restriction) have to be re-formulated for the chosen finite element space and mesh hierarchy. Our approach follows the *hardware-oriented numerics* paradigm and we aim at simultaneously maximising numerical and computational efficiency.

In this paper, we tackle the second problem and evaluate an implementation technique for geometric multigrid solvers that is based completely on sequences of sparse matrix-vector multiplications. With no loss in performance and only moderately increased memory requirements, this approach allows us to design a multigrid solver that is oblivious of the spatial dimension of the computational domain, the underlying unstructured discretisation grid, and the chosen finite element space. We are thus the first to assemble competitive geometric multigrid solvers for finite element discretisations on unstructured grids that execute very efficiently on both CPUs and GPUs.

Our numerical evaluation yields that the *FE-GMG* completely assembled by sequences of sparse matrix-vector kernels is able to exploit the parallelism provided by multicore CPUs and GPUs: We gain a speedup of two to three when doubling the amount of memory controllers of the CPU and yet another factor of four to 14 (eight in average) when switching from a multi-core CPU to the GPU. Additionally we show that the numbering of the degrees of freedom can have a huge impact on the performance, up to a factor of 25 on both architectures.

**Keywords:** GPGPU, unstructured grids, multigrid solvers, sparse matrices, finite elements.

## Abstract

We consider geometric multigrid solvers for linear systems stemming from the finite element discretisation of partial differential equations on unstructured grids. Our implementation technique reduces the complete solver to sequences of sparse matrix-vector multiplications and is thus well-suited for both GPUs and multicore CPUs. In particular, our implementation can handle several low- and high-order finite element spaces in 2D and 3D, while only the sparse matrix-vector kernel needs to receive significant tuning. For several benchmark problems, we achieve close to an order of magnitude speedup of a single GPU over multithreaded CPU code.

**Keywords:** GPGPU, unstructured grids, multigrid solvers, sparse matrices, finite elements.

# 1 Motivation and Introduction

Over the past several years, graphics processors (GPUs) have made the transition from an obscure (outside the domain of computer graphics, naturally) co-processor architecture to a valuable and widely accepted general purpose computing resource, both on standalone workstations and in large-scale HPC installations. For instance, the current (November 2010) leading machine in the TOP500 list of supercomputers, China's Tianhe-1A, derives its computational (2.5 PFLOP/s) and energy (4 MW) efficiency from employing more than 7 000 NVIDIA Fermi GPUs alongside its more than 14 000 Intel Nehalem quadcore CPUs. Despite significant improvements in compiler and tool-chain support, efficient GPU implementations are still challenging due to the fundamental architectural differences.

## 1.1 Current State of GPU Computing Research

The main reason why GPUs excel at many HPC workloads that provide ample parallelism is that their design is fundamentally different from commodity CPU architectures: Instead of minimising the latency of a single task, they maximise the overall throughput of a large set of tasks and consequently, the chip's ratio of functional units to control logic is much more favourable. For memory-bound problems, the GPU boards' more hard-wired memory lanes allow for a higher signal quality, and thus more aggregated memory bandwidth. We refer to a recent article by Garland and Kirk [1] for technical details and a concise description of the hardware-software model of *throughput-oriented computing*.

On the software and, more importantly in the scope of this paper, the algorithmic and data structure side, a lot of research efforts have been conducted. At first, the primary focus has been on re-designing implementations of single applications: As more and more application-specific success stories were told, more and more application scientists, in particular those with generally insatiable computational requirements, picked up on the trend and re-designed their codes to report significant improvements in the computational throughput ('time to solution') and energy efficiency. Some noteworthy examples include the coupling of real-time rendering with physics simulation for computer graphics and feature film, molecular dynamics simulations and medical imaging, to name just a few. Owens et al. and Garland et al. present surveys of such exemplary applications [2, 3]. Simultaneously, research into the parallelisation of fundamental computing primitives and data structures that are applicable for a wide range of applications has been intensified, and many (from the point of the average computational scientist, half-forgotten) techniques for inherently sequential operations and data access patterns have been re-discovered and re-evaluated for the new fine-grained parallel GPU architecture. A prominent example is the 'scan' primitive (parallel prefix sum), a technique that is not necessary in sequential code but which lies at the core of many building blocks like *sparse matrix-vector multiplication (SpMV)* (see Section 1.4), sorting and searching, solving recurrences etc. [4]. Similarly, both AMD and NVIDIA ship libraries for Fast-Fourier-Transformations and (dense) BLAS-like operations with their GPU computing toolkits, which are frequently updated with algorithmic and implementational improvements from the scientific community. Sparse (and of lesser interest in the scope of this paper, dense) linear system solvers have also received significant attention (see Section 1.2).

We are convinced that the latter category constitutes the more important consequence of the surge of GPU computing over the past several years: GPUs have pushed forward the fundamental paradigm shift towards exploiting on-chip fine-grained parallelism. This holds particularly true for application domains where previously, sequential approaches usually sufficed. This paper is thus not concerned with a single application but with an implementation technique of an important algorithm, (geometric) multigrid solvers for sparse linear systems, that can be applied in many different scenarios. Before we outline our approach in Section 1.5, we first discuss related work and the current state of the art in the context of this paper in the next three sub-sections.

## 1.2 Geometric and Algebraic Multigrid Methods for PDEs

*Multigrid (MG)* is *the* preferred numerical technique for solving large sparse linear systems of equations, in particular for those arising in the finite element (FE), finite difference and finite volume discretisation of PDEs where the condition number of the system matrix deteriorates with the mesh width $h$ and thus increasing problem size: In contrast to other iterative methods, *optimal* multigrid solvers converge independently of the grid width $h$ for a given problem with increasing refinement (decreasing $h$), and require only a linear amount of operations in the number of unknowns.

The theoretical foundation of multigrid methods is that high-frequency error components can be eliminated quickly using elementary iterative methods (like Jacobi or Gauß-Seidel), while low-frequency error components are more persistent. However, low-frequency errors on a fine mesh resolution (the so-called multigrid *level*) appear as high-frequency errors on a coarser mesh. Multigrid is thus an inherently recursive defect correction method acting on a hierarchy of different mesh resolutions: Starting on a fine mesh, the error is *smoothed* with a few iterations of an elementary iterative scheme. Then, the resulting defect is *restricted* to a coarser mesh, where the algorithm is recursively applied. At the coarsest mesh, the so-called *coarse grid problem* is solved exactly, and the resulting defect correction is *prolongated* to the next finer level where any remaining high-frequency errors are eliminated by *post-smoothing*. One such solver iteration is called a *multigrid cycle*, and different cycle types exist, depending on the order in which different mesh resolutions are traversed. It is worth noting that multigrid can be formulated non-recursively to improve the efficiency of a concrete implementation. For details, we refer to standard multigrid textbooks [5].

Two fundamentally different multigrid schemes exist: In *algebraic multigrid (AMG)*, the starting point is the discretised problem on the finest mesh, and coarser representations are constructed algebraically from the system matrix. In *geometric multigrid* which we employ throughout this paper, the starting point is the coarse mesh problem, which is then refined into a mesh hierarchy according to certain rules. For instance, in the $Q_i$ finite element spaces considered in this paper, each quadrilateral element at a given level is subdivided conformingly into four new quadrilaterals. However, geometric multigrid is not limited to such conforming subdivisions, extensions exist for hanging nodes during refinement, and for nonconforming (non-nested) mesh hierarchies. Even though AMG has beneficial black-box properties, we are convinced that geometric multigrid *combined* with finite element approaches (FE-GMG) is actually more advantageous from a numerical point of view. By including the underlying discretisation in the design of MG components, the numerical properties of the solvers can be optimised significantly and made much more robust, which outweighs the black-box character of AMG: The two components to target in this optimisation are the smoothing and grid transfer operations. For example, AMG can only identify grid anisotropies through strongly varying coefficients in the system matrix and eventually react with techniques like semi-coarsening, while the combined FE-GMG can employ true element evaluations to generate optimal (in terms of the approximation error) grid transfer operations, see Section 2.1. As a consequence, superconvergence effects can be achieved by combining geometric MG with higher-order finite element discretisations [6] in contrast to more black-box AMG techniques.

## 1.3 Multigrid Methods on GPUs

On GPUs, multigrid methods have only received moderate attention during the past several years, at least compared to the total number of papers concerned with sparse linear system solvers or finite element/volume/difference discretisations. The first to implement multigrid solvers for flow simulation in computer graphics entirely on GPUs were Bolz et al. and Goodnight et al. [7, 8]. They used GMG/AMG for finite-difference type discretisations on structured grids (in our opinion, AMG anf GMG essentially coincide for fixed-stencil multigrid, i. e., structured power-of-two grids and discretisations where unknowns coincide with grid points). More recent publications presenting applications that require multigrid solvers are supersonic flows (AMG, unstructured grids [9]), (interactive) flow simulations for feature film (AMG/GMG, structured [10, 11]), out-of core multigrid for gigapixel image stitching (GMG/AMG, structured [12]), image denoising and optical flow (GMG/AMG, structured [13]), power grid analysis (AMG, structured/unstructured [14]) and blood flow in the human heart (AMG, unstructured [15]). This last paper is at first sight similar in spirit to our work, since the authors also reduce (almost) the entire multigrid algorithm to sequences of sparse matrix-vector multiplications. The important difference (besides AMG vs. GMG) is that they use a fixed, problem-specific data layout in their SpMV implementation whereas we go further and, e. g., use layouts that have been shown to deliver superior performance for a wide range of non-zero patterns, see also Section 1.5. In summary, we can say that previous publications describing multigrid on GPUs either target algebraic multigrid, or are limited to structured grid geometric multigrid and low-order discretisations. To the best of our knowledge, we are the first to present *geometric MG for high-order FEM on GPUs*.

Finally, we want to point out that ‘multigrid’ alone is not a good indicator that state-of-the-art numerics is being used (even though it is certainly superior to Krylov subspace methods). The reason for this is that the convergence of geometric and algebraic multigrid alike strongly depends on the quality of the smoothing operation. ‘Standard’ smoothing operators like Jacobi and Gauß-Seidel exhibit severe difficulties in the case of nonlinearities and anisotropic meshes and operators. Commonly known, numerically favourable and ‘strong’ smoothers like ILU($k$) typically exhibit highly recursive, inherently sequential data dependencies. To the best of our knowledge, no such method has been presented on GPUs so far, at least not in the unstructured case. Some of the authors of this paper have previously worked on geometric multigrid solvers equipped with strong smoothers that exploit locally structured grids [16, 17]. Throughout this paper, we employ simple Jacobi smoothing and thus only tackle one of the two components in unstructured grid FE-GMG that are required for numerically and computationally optimised solvers on GPUs. In Section 5 we briefly discuss approaches to tackle the second open problem, namely stronger smoothers for unstructured grids. Since the numerical properties of stronger smoothers may depend on the numbering scheme of the unknowns, our evaluation of the impact of numbering schemes only highlights architectural and hardware influence. This separation of hardware and numerical aspects is only possible with simple smoothers.

## 1.4 Sparse Matrix-Vector Multiplication

The key component in our GPU implementation of multigrid for different finite element discretisations is an efficient parallel sparse matrix-vector multiplication kernel. Implementations for multicore CPUs and the Cell BE processor, including a discussion of better data layouts compared to the 'standard' *compressed sparse row (CSR)* format, have been presented by Williams et al. and Goumas et al. [18, 19]. Buatois et al. [20] were the first to discuss the performance of SpMV on modern GPUs, they employed a *block-CSR* format which exploits locally dense sub-blocks in the matrix to improve efficiency. Bell and Garland [21] and Baskaran and Bordawekar [22] evaluated different storage formats like the ELLPACK/ITPACK format [23], the coordinate format COO, and vector-CSR formats. They concluded that ELLPACK is the best-suited general purpose format, i. e., it delivers consistently high performance for a wide range of sparsity patterns. This work has recently been extended by Vazquez et al. [24] to a new format, ELLPACK-R, which stores an additional array to allow for a more efficient implementation. They propose an entire family of implementations based on the number $T$ of threads assigned to each matrix row, called ELLR-T. This parameter allows for applying autotuning techniques, and to the best of our knowledge, this is the fastest currently available general purpose SpMV implementation on GPUs. Our implementation is based on the freely available code by Bell and Garland, but has been extended to the ELLPACK-R format and thus implements some of the improvements suggested by Vazquez et al. Finally, we refer to a book chapter by Williams et al. [25] for a survey on the state of the art of SpMV on multicore CPUs, GPUs and the Cell BE processor.

## 1.5 Paper Contribution and Overview

In this paper, we evaluate the performance of an implementation technique for FE-GMG (finite element geometric multigrid) solvers for PDE problems discretised on unstructured grids. Our target architectures are medium-grained parallel multicore CPUs and fine-grained (manycore) GPUs. At this point, we focus entirely on the solver performance, evaluating it for different unstructured grids and finite element spaces. This is justified since in many practical scenarios, the linear solver often dominates the total execution time.

The solver is implemented (almost) entirely as sequences of sparse matrix-vector multiplications. This admittedly simple idea has surprisingly many beneficial properties: The multigrid solver needs to be implemented only once, and is completely oblivious of the underlying finite element space, and even oblivious of the dimension of the computational domain (2D, 3D). Furthermore, our implementation replaces many specialised kernels (for which the development of fine-grained parallelisation techniques has only very recently begun, see Section 5) with one central, well-understood and well-optimised parallel kernel, which is favourable in terms of software maintainability and the rapid adoption of GPUs in multigrid and finite element codes.

It should be noted that our approach suffers almost no performance penalty for *unstructured* grids, compared to implementing prolongation and restriction specifically tailored to the underlying finite element space. Of course, the underlying discretisation can only be hidden from the solver if other stages of the solution process are more dependent on the discretisation. Since the system matrices and the right hand sides have to be assembled prior to executing the linear solver anyway, we argue that the assembly of the grid transfer matrices can be realised in this stage as well at minimum additional cost.

The remainder of this paper is structured as follows: In Section 1.1, we have argued that GPU computing has evolved into an established technique. We therefore refrain from presenting yet another primer of the GPU programming model, and instead refer the interested reader to the above mentioned survey articles and the references therein [1, 2, 3]. In Section 2 we provide the necessary mathematical background of our approach, in particular on how to efficiently map grid transfer operations to sparse matrix-vector multiplications. Section 3 covers implementational details, and in Section 4 we present numerical experiments highlighting the benefits of our approach and the speedup obtained by a GPU over carefully tuned multithreaded CPU code. Here, we also analyse the importance of choosing a suitable ordering scheme for the unknowns, and its consequences for solver performance. We conclude with avenues for future work in Section 5.

# 2 Multigrid Solver Components

## 2.1 Geometric Multigrid as a Sequence of SpMV

Geometric multigrid algorithms comprise the following components, which in our implementation are mapped to individual kernels. We refer to standard multigrid textbooks for details on the control flow logic, in particular on the cycle control.

- Defect calculations are realised as a straight forward extension of sparse matrix-vector multiplication, i. e., we do not transfer the coefficient vector back and forth to off-chip memory, but rather fuse the SpMV and vector-vector operations into a single kernel. We implement smoothing as a short defect correction loop, so defect calculations (and hence, SpMV) are required once in each pre- and postsmoothing step on each level of the mesh hierarchy. As mentioned in Section 1.3, we currently limit ourselves to simple Jacobi smoothing, which is realised by a straight forward vector-vector operation.

- Grid transfers, i. e., prolongations and restrictions depending on the chosen finite element spaces and grid hierarchies, are written as sparse matrix-vector multiplications, see Section 2.3. We use two different sets of matrices to avoid having to rewrite the SpMV kernel to perform operations with the transpose of the matrix.

- The coarse grid solver generally requires very little time, and we use a preconditioned conjugate gradient solver for this task, which in turn is, once SpMV and the BLAS-1 operations DOT and NRM2 are available, trivial to implement on both CPUs and GPUs. Finally, the coarse grid correction after restriction is a simple vector-vector operation for conforming grids. In the nonconforming case, the computation of an optimal steplength parameter is in turn realised by SpMV and few dot product operations.

From this list, it is obvious that indeed, the vast majority of arithmetic operations in the multigrid cycle are based on our SpMV kernels, and the computational efficiency and performance of the SpMV kernel should be transferred directly to the entire solver. We assess this claim in Section 4.4.

## 2.2 Ordering Techniques for Degrees of Freedom

It is well known that even for the same finite element discretisation on the same mesh, runtime performance may vary considerably depending on the ordering/numbering of the degrees of freedom. For simple Jacobi smoothing, the numerical performance (rate of convergence) is independent of the ordering, but for more advanced smoothers, the convergence rate is influenced substantially by the underlying ordering [26]. We implemented a simple preprocessing routine that allows us to convert between different numbering schemes, and include the following five variants in our benchmark in Section 4. This allows us to not only identify the 'best' ordering scheme for a given problem, but more importantly it significantly increases the range of experiments we perform without adding additional (semi-artificial) test geometries and coarse grids. The analysis of the numerical impact of different numbering schemes will be addressed in a future publication targeting stronger smoothers in our setting, see Section 5.

In two-level numbering, **2LV**, the numbers of the degrees of freedom on each finer mesh coincide with the numbers on the corresponding coarser mesh. The popular Cuthill McKee numbering, **CM**, is designed to reduce the matrix' bandwidth. In the **XYZ** technique, degrees of freedom are sorted according to their spatial coordinates, using the $x$-coordinate as leading dimension. In order to simulate fully adaptive meshes (hence fully unstructured matrices) we employ a randomly permutated numbering, **STO**, that maximises cache miss rates. This numbering scheme is of course artificial and only included as a worst-case reference. Finally, a hierarchical approach **HIE** is used, where the degrees of freedom of cells in the fine grid are recursively collected and numbered according to the coarse grid cells.

## 2.3 Grid Transfers in Multigrid

In this paper, we restrict ourselves to the case where, for $d, k \geq 1$, $V_{2h}$ and $V_h$ are both conforming $Q_k$ finite element spaces [27] defined on unstructured conforming $d$-dimensional hypercube grids $\Omega_{2h}$ and $\Omega_h$, where $\Omega_h$ is the grid refined from $\Omega_{2h}$ by subdividing each $d$-dimensional hypercube into $2^d$ children. Since $V_{2h} \subset V_h$, we can perform a grid transfer of any $u_{2h} \in V_{2h}$ into $V_h$ by interpolating $u_{2h}$.

In the following, we show that, if we choose the standard Lagrange bases for $V_{2h}$ and $V_h$, the interpolation, and therefore the grid transfer, can be expressed as a sparse matrix-vector product. We note that the same approach can be used for simplex-based grids and the $P_k$ family of conforming finite element discretisations.

Let $\varphi_h^{(1)}, \ldots, \varphi_h^{(m)}$ be the standard Lagrange basis of $V_h$, then the interpolant $u_h$ of a $u_{2h} \in V_{2h}$ can be calculated by evaluating $u_{2h}$ in the corresponding nodal points $\xi_h^{(i)}$ of $\varphi_h^{(i)}$:

$$\begin{aligned} u_h &:= \sum_{i=1}^{m} x_i \cdot \varphi_h^{(i)} \qquad (1) \\ x_i &:= u_{2h}(\xi_h^{(i)}). \end{aligned}$$

Now, for a set of basis functions $\varphi_{2h}^{(1)}, \ldots, \varphi_{2h}^{(n)}$ of $V_{2h}$ and a $u_{2h} \in V_{2h}$ given by

$$u_{2h} = \sum_{j=1}^{n} y_j \cdot \varphi_{2h}^{(j)},$$

with a coefficient vector $y \in \mathbb{R}^n$, we can express (1) as

$$\begin{aligned} u_h &:= \sum_{i=1}^{m} x_i \cdot \varphi_h^{(i)} \\ x &:= P_{2h}^{h} \cdot y \qquad (2) \end{aligned}$$

where the $m \times n$ prolongation matrix $P_{2h}^h$ is given by

$$(P_{2h}^h)_{ij} = \varphi_{2h}^{(j)}(\xi_h^{(i)}). \qquad (3)$$

Additionally, if $\varphi_{2h}^{(1)}, \ldots, \varphi_{2h}^{(n)}$ is the standard Lagrange basis, it holds that for any $\xi \in \Omega$:

$$\left| \left\{ \varphi_{2h}^{(j)} \mid \varphi_{2h}^{(j)}(\xi) \neq 0;\ 1 \leq j \leq n \right\} \right| \leq (k+1)^d,$$

so the prolongation matrix (3) has at most $(k+1)^d$ non-zero entries per row and the grid transfer can be realised efficiently as the sparse matrix-vector product (2).

The matrix $A_h$ representing the discrete Laplace operator on $V_h$ has at least $(k+1)^d$ non-zero entries per row while having the same number of rows as $P_{2h}^h$, so the additional amount of memory necessary to store the prolongation matrix $P_{2h}^h$ and, analogously the restriction matrix $\left(P_{2h}^h\right)^\top$ is bounded by the memory requirements of $A_h$. This is just another example of trying to achieve better performance at the cost of moderately increased memory requirements, since the alternative approach to implement grid transfers directly (i. e., without full assembly of the prolongation matrices) yields no performance improvement in our experience.

### 2.3.1 Example: Prolongation Matrix for 2D $Q_1$

In the case where $V_h$ and $V_{2h}$ are 2D $Q_1$ discretisations, the nodal points $\xi_h^{(i)}$ of $V_h$ coincide with the vertices $v_h^{(i)}$ of the grid $\Omega_h$. Each vertex $v_h^{(i)}$ of $\Omega_h$ corresponds to either a vertex of $\Omega_{2h}$ or to the midpoint of an edge or a quadrilateral of $\Omega_{2h}$. Let $n_{2h}^v$ denote the total number of vertices, $n_{2h}^e$ the number of edges and $n_{2h}^q$ the number of quadrilaterals of $\Omega_{2h}$, then the total number of vertices of $\Omega_h$ is given as $n_h^v = n_{2h}^v + n_{2h}^e + n_{2h}^q$. By choosing an appropriate 2-level-numbering scheme of the vertices $v_h^{(i)}$ of $\Omega_h$ (see Section 2.2), and therefore the basis functions $\varphi_h^{(i)}$ of $V_h$, the $n_h^v \times n_{2h}^v$ prolongation matrix $P_{2h}^h$ has the block structure

$$P_{2h}^h = \begin{bmatrix} P_v \\ P_e \\ P_q \end{bmatrix},$$

where $P_v$ is an $n_{2h}^v \times n_{2h}^v$ identity matrix, $P_e$ is an $n_{2h}^e \times n_{2h}^v$ and $P_q$ an $n_{2h}^e \times n_{2h}^v$ matrix. Each row $i$ of $P_e$ represents the vertex of the grid $\Omega_h$ which corresponds to the midpoint of edge $i$ of $\Omega_{2h}$. Therefore, each row of $P_e$ has exactly two non-zero entries, each storing the value $\frac{1}{2}$, where the column indices correspond to the edge's corner vertex indices. In analogy to $P_e$, $P_q$ represents the vertices corresponding to quadrilateral midpoints of $\Omega_{2h}$, so each row of $P_q$ has four non-zero entries storing the value $\frac{1}{4}$, where the column indices correspond to the indices of the quadrilateral's corner vertices.

# 3 Implementation for Multicore CPUs and GPUs

Our multigrid solver is implemented on top of our HONEI libraries [28], which provide all necessary infrastructure for parallelisation and have been extended significantly since the original publication. HONEI abstracts from the target hardware, i. e., the target architecture is a template parameter: The exact same multigrid solver code has only been implemented once, and the appropriate compute kernels for the multicore and the GPU implementation are inserted at compile-time. This approach makes the code well maintainably and extendably. The GPU implementation provides support for recent NVIDIA GPUs and has been realised as a wrapper around CUDA [29]. As our solver is almost exclusively built around calls to highly efficient SpMV kernels (see Section 2.1), it suffices to describe our approach to parallelising and tuning the SpMV kernel.

## 3.1 The ELLPACK-R Format for Sparse Matrices

We do not utilise the 'standard' CSR format, but rather the ELLPACK-R format (an extension to the ELLPACK/ITPACK format [23]) proposed by Vazques et al. [24], see Section 1.4. In our experience, ELLPACK(-R) leads to significantly higher computational throughput, even for sequential code (see also Section 1.4 and 4.3).

This format stores the sparse matrix $S$ in two arrays `A` and `j`. `A` stores the non-zero entries of $S$ in column-major order in an array of size `RowCount` × `MaxRowSize`, where `RowCount` is the number of rows in $S$ and `MaxRowSize` is the maximum number of non-zero entries in any row of $S$: All shorter rows in the matrix `A` are appropriately padded with zeros, resulting in a equal size of every row. The array `j` holds the column index of every entry in `A`. In addition to this common ELLPACK data structure, the ELLPACK-R format adds one additional array `rl` of size `RowCount`. The array stores the effective count of non-zero elements on every row without the padded zero elements. This enables an algorithm to stop computation on a row when all non-zero elements have been processed. A small illustrative example is:

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \quad \Rightarrow \quad \mathtt{A} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \mathtt{j} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix} \mathtt{rl} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 2 \end{bmatrix}$$

## 3.2 Sparse Matrix-Vector Multiplication

Based on the ELLPACK-R format, the sparse matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$ can be performed by computing each entry $y_i$ of the result vector $\mathbf{y}$ independently:

$$y_i = \sum_{nz=0}^{rl_i} A_{i,nz} \times x_{j_{nz}}.$$

In general, this results in a comparatively regular access pattern on the data of $\mathbf{y}$ and $A$. In contrast, the access pattern on $\mathbf{x}$ depends highly on the non-zero structure of $A$.

### 3.2.1 GPU Implementation

The ELLPACK-R based SpMV kernel is mapped to the GPU architecture by launching one device thread for the calculation of an entry $y_i$. This approach has the following advantages: The data access on the arrays three ELLPACK-R arrays and to $\mathbf{y}$ is fully coalesced due to the column-major ordering used. The access to the array $\mathbf{x}$ can be cached via the texture cache on the GPU to improve efficiency. On the FERMI generation of GPUs (currently unfortunately not available to us), the device-wide L2-cache is also well utilised. No synchronisation between threads is necessary. The threads in one CUDA warp can not diverge because no flow instruction is used that would cause serialisation. Every warp finishes execution directly when all non-zero entries in its threads' rows are completely processed. Because of this, only warps with a high relative non-zero count in their rows execute longer compared to average warps.

### 3.2.2 CPU Implementation

The CPU implementation follows essentially the same lines as the GPU implementation. Instead of launching `RowCount` threads, we launch one thread per CPU core available to us, resulting in a scalable implementation.

In detail, if four threads are used for the calculation, the vector $\mathbf{y}$ is virtually partitioned in four contiguous parts. Then every thread computes the entries $y_i$ of his part of $\mathbf{y}$ sequentially, but independently of the other threads. The arrays of the matrix and the vector $\mathbf{x}$ stay in the shared main memory and are accessed by all threads read-only. As outlined before, the data access pattern on the ELLPACK-R arrays and on $\mathbf{y}$ is very regular, supporting the processor's internal cache prefetching units. On the other hand, the locality of the data access on the array $\mathbf{x}$, although not predictable, utilises the whole cache hierarchy of modern CPUs, including the last level cache, shared by all CPU cores. The underlying multicore backend uses a static thread pool as described by Mallach [30], which allows for fast launching and terminating of threads. Thus, the amount of threads need not be fixed to the CPU core count but can be adapted to varying sizes of the array $\mathbf{x}$. The thread pool also ensures proper thread pinning and memory affinity on NUMA systems.

# 4 Results and Performance Experiments

As motivated in Section 1.5, we concentrate on the *solution* stage of the FE-GMG solver and pre-assemble all necessary matrices and vectors, including the transfer matrices. Thus, only the execution time and corresponding speedups between architectures and/or sorting methods are measured.

All benchmarks are performed on an Intel Core i7 920 quadcore workstation including an NVIDIA GeForce GTX 285 GPU. The dual memory-controller design of the i7 offers 33 GByte/s bandwidth to off-chip memory and the GTX 285 delivers 160 GByte/s, respectively. The CPU features 8 MB L3 cache shared between all cores and 256 kB L2 cache for each core, while the GPU only has 16 kB of texture cache, the same amount as the L1 cache for each core on the CPU.

## 4.1 Numerical Accuracy

In all subsequent benchmarks, we made sure that all numerical results are as accurate as if computed by the finite element package FEAT2/FEATFLOW that we used to assemble the matrices and vectors: The multigrid solvers exhibit the same convergence behaviour, and the very small differences in the result vectors are due to floating point noise since the order of operations changes in the parallel implementation.

## 4.2 Benchmarks and Solver Setups

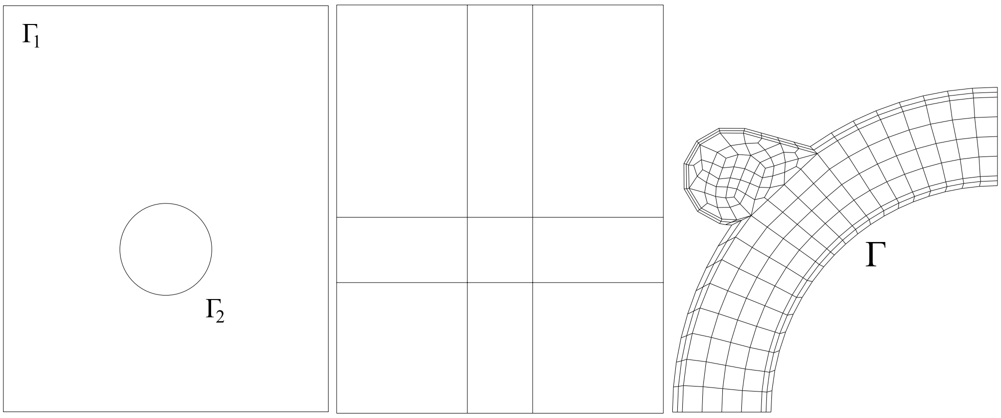
In order to show that all speedups are gained mesh-independently, we employ two different test meshes: In the CIRCLE setup, the computational domain is rectangular, enclosing an inner circle, see Figures 1 (a) and (b) which depict the geometry and the coarse mesh of the setup. A more complex coarse mesh is used within the BLOOD benchmark, where the domain consists of an idealised blood vessel with an aneurysm [31], the coarse mesh is depicted in Figure 1 (c). This setup is taken from a fluid-structure interaction benchmark, the finely resolved boundary layer is a discretisation of the solid part (the blood vessel's walls) and not part of the fluid domain. In both cases, we solve the Poisson problem and employ Dirichlet boundary conditions on the outer (and inner) boundaries $\Gamma$, $\Gamma_1$ and $\Gamma_2$, respectively:

$$\text{CIRCLE: } \begin{cases} -\Delta u = 1, & \mathbf{x} \in \Omega \\ u = 0, & \mathbf{x} \in \Gamma_1 \\ u = 1, & \mathbf{x} \in \Gamma_2 \end{cases} \quad \text{BLOOD: } \begin{cases} -\Delta u = 0, & \mathbf{x} \in \Omega \\ u = 1, & \mathbf{x} \in \Gamma \end{cases}$$

The Poisson problem is the most fundamental (elliptic) PDE. It arises for instance in electrostatics (potential calculations), in structural mechanics (linearised elasticity with small deformations), and as a sub-problem in many fluid flow simulations (Pressure-Poisson problem in operator-splitting approaches for the Navier-Stokes equations). Furthermore, the Poisson problem is a very convenient model problem not only for elliptic PDEs since it is difficult to solve, its discrete variant can be made arbitrarily ill-conditioned, and few problem-specific optimisations can be applied.

We configure our multigrid solver to perform a V-cycle always traversing the full mesh hierarchy. Coarse grid problems are treated with a Conjugate Gradient solver using Jacobi preconditioning, it is configured to reduce the initial residual by two digits. However, the smoothing parameters of the multigrid have to be configured differently for the two problems: In order to ensure convergence with the number of iterations not depending on the refinement level in case of the BLOOD setup, significantly more pre- and postsmoothing steps are required and the damping parameter has to be amplified differently: We use four pre- and postsmoothing steps for the CIRCLE setup and 64 for BLOOD. The damping parameter of the Jacobi smoother is set to $0.7$ and $0.5$, respectively. This is a simple consequence of the anisotropies present in the BLOOD problem since we are currently limited to Jacobi smoothing. In other words, this problem requires a much stronger smoother and needs to be re-evaluated in future work. All benchmarks are performed in double precision.

Table 1 contains refinement levels and numbers of degrees of freedom for the two benchmark configurations and the two FE spaces covered in our evaluation. The maximum level of refinement is chosen so that the largest possible fine mesh when using $Q_2$ finite elements just barely fits into the 2 GB GPU memory, for the two problems respectively.



(a) geometry of the CIRCLE setup (b) coarse grid of the CIRCLE setup (c) coarse grid of the BLOOD setup

Figure 1: Benchmark setups CIRCLE and BLOOD

| | CIRCLE | | | | BLOOD | | | |
|---|---|---|---|---|---|---|---|---|
| | $Q_1$ | | $Q_2$ | | $Q_1$ | | $Q_2$ | |
| L | N | non-zeros | N | non-zeros | N | non-zeros | N | non-zeros |
| 4 | 576 | 4552 | 2176 | 32192 | 15233 | 133890 | 60417 | 1205010 |
| 5 | 2176 | 18208 | 8448 | 128768 | 60417 | 535561 | 240641 | 4820049 |
| 6 | 8448 | 72832 | 33280 | 515072 | 240641 | 2149385 | 960513 | 19344465 |
| 7 | 33280 | 291328 | 132096 | 2078720 | 960513 | 8611849 | 3837953 | 77506641 |
| 8 | 132096 | 1172480 | 526336 | 8351744 | 3837953 | 34476041 | - | - |
| 9 | 526336 | 4704256 | 2101248 | 33480704 | - | - | - | - |
| 10 | 2101248 | 18845696 | - | - | - | - | - | - |

Table 1: Refinement levels (L), corresponding numbers of degrees of freedom (N) and numbers of non-zeros of system matrices

## 4.3 Performance of the SpMV Kernel

We first demonstrate the efficiency of the applied SpMV kernel for our modified ELLPACK matrix format since it is the major component of our multigrid solver. Table 2 provides MFlop/s rates for the singlethreaded CPU (SSE) version, the multicore implementation using an optimal number of four threads, and the CUDA kernel. The matrices are taken from the linear systems of the CIRCLE setup on the various levels of refinement. The tables also show the numbers of non-zero elements on each level. Multicore speedups are calculated vs. the singlecore variant, and GPU speedups are calculated vs. the multicore variant. We thus present a very 'honest' full 'chip vs. chip' comparison.

In general, the SSE backend performs with 500 MFlop/s at most, which is in line with recent results for optimised SpMV kernels on commodity CPUs [18]. However, the performance is severely dropping when using the **STO** numbering technique, as expected. Using the full multicore capacity of the i7 system, the performance level is virtually doubled in average (sometimes tripled), which clearly demonstrates that the SpMV kernel scales rather with the number of memory controllers than with the number of CPU cores for sufficiently large input matrices. As a side note, we have also executed all tests with eight CPU threads via HyperThreading, and found that the performance is surprisingly on par with employing four native threads: This again

|  |  |  | $Q_1$ |  |  |  |  | $Q_2$ |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| L | SSE | MCSSE | speedup | CUDA | speedup | SSE | MCSSE | speedup | CUDA | speedup |
| 6 | - | - | - | - | - | 492 | 894 | 1.82 | 7985 | 8.93 |
| 7 | 870.81 | 2445.06 | 2.81 | 7441.65 | 3.04 | 467 | 612 | 1.31 | 8527 | 13.93 |
| 8 | 672.14 | 1163.98 | 1.73 | 8411.42 | 7.23 | 358 | 569 | 1.59 | 7942 | 13.96 |
| 9 | 506.85 | 988.19 | 1.95 | 7928.9 | 8.02 | 323 | 528 | 1.63 | 7680 | 14.55 |
| 10 | 426.81 | 855.99 | 2.01 | 7925.34 | 9.26 | - | - | - | - | - |
| 6 | - | - | - | - | - | 479 | 933 | 1.95 | 6168 | 6.61 |
| 7 | 790 | 2897 | 3.67 | 7247 | 2.5 | 458 | 883 | 1.93 | 7035 | 7.97 |
| 8 | 685 | 1268 | 1.85 | 8459 | 6.67 | 289 | 802 | 2.78 | 6470 | 8.07 |
| 9 | 445 | 1187 | 2.67 | 7539 | 6.35 | 262 | 743 | 2.84 | 6288 | 8.46 |
| 10 | 399 | 1120 | 2.81 | 7314 | 6.53 | - | - | - | - | - |
| 6 | - | - | - | - | - | 500 | 1096 | 2.19 | 6706 | 6.12 |
| 7 | 842 | 3299 | 3.92 | 8506 | 2.58 | 491 | 950 | 1.93 | 7677 | 8.08 |
| 8 | 760 | 1344 | 1.77 | 10403 | 7.74 | 334 | 897 | 2.69 | 7911 | 8.82 |
| 9 | 504 | 1369 | 2.72 | 11007 | 8.04 | 330 | 836 | 2.53 | 8074 | 9.66 |
| 10 | 494 | 1372 | 2.78 | 11176 | 8.15 | - | - | - | - | - |
| 6 | - | - | - | - | - | 416 | 841 | 2.02 | 5057 | 6.01 |
| 7 | 697 | 2048 | 2.94 | 5880 | 2.87 | 346 | 787 | 2.27 | 3820 | 4.85 |
| 8 | 497 | 981 | 1.97 | 4257 | 4.34 | 244 | 590 | 2.42 | 2468 | 4.18 |
| 9 | 348 | 843 | 2.42 | 2628 | 3.12 | 160 | 366 | 2.29 | 1689 | 4.61 |
| 10 | 224 | 443 | 1.98 | 1794 | 4.05 | - | - | - | - | - |
| 6 | - | - | - | - | - | 487 | 911 | 1.87 | 6454 | 7.08 |
| 7 | 809 | 3148 | 3.89 | 8049 | 2.56 | 482 | 852 | 1.77 | 7465 | 8.76 |
| 8 | 738 | 1313 | 1.78 | 9726 | 7.41 | 300 | 836 | 2.79 | 7776 | 9.3 |
| 9 | 471 | 1345 | 2.86 | 10342 | 7.69 | 299 | 782 | 2.62 | 7903 | 10.11 |
| 10 | 465 | 1331 | 2.86 | 10553 | 7.93 | - | - | - | - | - |

Table 2: Performance of SpMV (Mflop/s) for $Q_1$ (left) and $Q_2$ (right). From top to bottom: **2LV**, **CM**, **XYZ**, **STO** and **HIE** numbering schemes

underlines the impact of the number of memory controllers on the Nehalem/i7 CPU architecture. Finally, the GPU can exploit its bandwidth-advantage and provides up to 14 GFlop/s, 8 GFlop/s in general when taking **STO** out of the calculation, delivering an additional speedup of eight compared to the fully exploited i7 chip. We address the differences in performance when switching the ordering technique separately in Section 4.5.

## 4.4 Performance of the Full Solver

We exemplarily provide total execution times and speedups for the multigrid solver corresponding to the SpMV benchmarks of the CIRCLE system matrices in the last sub-section in Table 3 and speedups only for the BLOOD setup in Table 4.

It can be seen that the speedups projected by the SpMV benchmark in Section 4.3 translate very well to the solver level in our approach. In absolute numbers, our results mean that computations consuming tens of seconds of time on a fully exploited multicore CPU can be accomplished by the GPU in one or two seconds. Finally, these findings are problem-independent, as the speedups are consistent between the two problems, see Table 4 for the speedup data.

## 4.5 Impact of the Numbering Scheme for the Two Example Problems

With respect to the specific numbering in use, we find that a preferred order of the techniques can be found independently of the hardware architecture: While **STO** always performs worst, the other techniques, **CM**, **HIE**, and **2LV** speed up the computations by factors of approximately 7, 1.5 and another 1.5 respectively. The **XYZ** performs

|  |  |  | $Q_1$ |  |  |  |  | $Q_2$ |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| L | SSE | MCSSE | speedup | CUDA | speedup | SSE | MCSSE | speedup | CUDA | speedup |
| 6 | - | - | - | - | - | 0.6 | 0.53 | 1.14 | 0.16 | 3.31 |
| 7 | 0.17 | 0.16 | 1.03 | 0.08 | 2.05 | 2.58 | 2.08 | 1.24 | 0.34 | 6.12 |
| 8 | 0.75 | 0.62 | 1.2 | 0.14 | 4.34 | 11.99 | 8.65 | 1.39 | 0.91 | 9.51 |
| 9 | 3.87 | 2.63 | 1.47 | 0.38 | 7.01 | 56.72 | 33.72 | 1.68 | 3.22 | 10.47 |
| 10 | 18.46 | 10.02 | 1.84 | 1.27 | 7.92 | - | - | - | - | - |
| 6 | - | - | - | - | - | 0.54 | 0.33 | 1.64 | 0.15 | 2.14 |
| 7 | 0.2 | 0.62 | 0.32 | 0.14 | 4.43 | 2.39 | 1.36 | 1.76 | 0.33 | 4.13 |
| 8 | 1.03 | 0.76 | 1.35 | 0.2 | 3.88 | 14.6 | 6.1 | 2.39 | 0.99 | 6.13 |
| 9 | 5.79 | 2.87 | 2.02 | 0.52 | 5.52 | 73.99 | 25.57 | 2.89 | 3.8 | 6.72 |
| 10 | 27.27 | 12.39 | 2.2 | 1.86 | 6.66 | - | - | - | - | - |
| 6 | - | - | - | - | - | 0.47 | 0.31 | 1.52 | 0.14 | 2.23 |
| 7 | 0.14 | 0.38 | 0.37 | 0.13 | 2.96 | 2.02 | 1.15 | 1.76 | 0.29 | 3.92 |
| 8 | 0.75 | 0.58 | 1.29 | 0.15 | 3.86 | 11.31 | 4.59 | 2.47 | 0.79 | 5.8 |
| 9 | 4.04 | 2.13 | 1.9 | 0.34 | 6.29 | 47.05 | 20.15 | 2.33 | 2.7 | 7.45 |
| 10 | 16.96 | 8.14 | 2.08 | 1.07 | 7.64 | - | - | - | - | - |
| 6 | - | - | - | - | - | 3.16 | 1.87 | 1.69 | 0.72 | 2.6 |
| 7 | 1.16 | 1.3 | 0.89 | 0.48 | 2.72 | 15.68 | 7.35 | 2.13 | 1.94 | 3.79 |
| 8 | 6.19 | 4.72 | 1.31 | 1.15 | 4.1 | 82.92 | 36.26 | 2.29 | 8.85 | 4.1 |
| 9 | 34.83 | 17.02 | 2.05 | 4.85 | 3.51 | 483.92 | 221.8 | 2.18 | 47.02 | 4.72 |
| 10 | 204.84 | 107.34 | 1.91 | 25.23 | 4.25 | - | - | - | - | - |
| 6 | - | - | - | - | - | 0.55 | 0.43 | 1.28 | 0.16 | 2.72 |
| 7 | 0.18 | 0.19 | 0.95 | 0.08 | 2.23 | 2.2 | 1.29 | 1.71 | 0.31 | 4.19 |
| 8 | 0.87 | 0.68 | 1.28 | 0.17 | 4.02 | 13.62 | 5.77 | 2.36 | 0.86 | 6.7 |
| 9 | 4.95 | 2.42 | 2.05 | 0.41 | 5.95 | 63.26 | 23.19 | 2.73 | 2.95 | 7.87 |
| 10 | 23.05 | 9.79 | 2.36 | 1.3 | 7.55 | - | - | - | - | - |

Table 3: Performance of Multigrid (execution time in seconds) for $Q_1$ (left) and $Q_2$ (right) and the CIRCLE setup. From top to bottom: **2LV**, **CM**, **XYZ**, **STO** and **HIE** numbering schemes

|  | speedup |  |  |  |
|---|---|---|---|---|
|  | $Q_1$ |  | $Q_2$ |  |
| numbering | MCSSE | CUDA | MCSSE | CUDA |
| 2LV | 2,16 | 6,65 | 2,3 | 7,4 |
| CM | 2,16 | 7,83 | 2,52 | 7,87 |
| XYZ | 2,19 | 6,81 | 2,05 | 6,38 |
| Stoch | 2,15 | 6,22 | 2,06 | 7,93 |
| Hie | 2,06 | 9,67 | 3,05 | 7,94 |

Table 4: multigrid speedups for largest problem size, BLOOD setup

best in all our benchmarks and delivers yet another factor of 1.5. The effect of a randomly created (or - in other words - *arbitrarily* created) sparsity pattern seems to be architecture-dependent: Taking a look at the speedup of one of the well-performing orderings compared with the **STO** distribution, the GPU performance benefits slightly more from a better sorting strategy than the CPU. These findings strongly underline that the effect of renumbering degrees of freedom alone can be critical and that performance can be increased up to a factor of 25 by employing reasonable orderings.

# 5 Conclusions and Future Work

We are convinced that the performance and speedup results we have presented are very promising: By using an implementation technique that exploits sparse matrix-vector multiplications whenever possible, we are the first to tackle one of the two fundamental problems preventing efficient, flexible and numerically highly optimised finite element geometric multigrid solvers for PDE problems on unstructured grids. Already at this intermediate stage, our solver is competitive for a wide range of practically relevant low- and high-order finite element spaces. In particular, we have demonstrated that the recent improvements of SpMV enable competitive implementations of multigrid solvers on GPUs for unstructured grids, whereas only two years ago, similar levels of speedup have only been possible in the structured and block-structured case.

Our approach opens up three main avenues for future work that we are currently pursuing actively: Firstly, the solver can be made more robust by enhancing it with stronger smoothers. Several examples exist of strong smoothers whose *application* can be written in the form of a sparse matrix-vector multiplication, and we postulate that such an addition would not result in a reduction of the speedup. The most prominent example of such approaches are smoothers of *sparse approximate inverse (SPAI)* type that have previously been demonstrated to be advantageous in geometric multigrid settings [32] compared to using standard smoothers, but in general are not as powerful as sequential ILU-type approaches. Closely coupled with designing stronger parallel smoothers is the question of finding optimal ordering techniques, and we plan to thoroughly investigate their impact. So far, their influence has been significant, but purely hardware-induced, while then, we will have to face the added level of complexity that not only the runtime, but also the numerical performance is (potentially inversely) affected by the numbering [26].

While in many cases it is justified to focus on accelerating the linear solver alone, simply because it often dominates the total time to solution, this approach to hybrid CPU-GPU computing is in the long run limited in speedup due to Amdahl's law. Consequently, also the assembly process needs to be pushed to the GPU. Cecka et al. and Komatisch et al. have recently investigated finite element assembly for unstructured grids on GPUs [33, 34]. We are convinced that our approach to 'pre-assemble' the discrete grid transfer operations can be tackled with similar techniques like the ones suggested for assembling the system matrix and the right hand side, and in the long run, also of smoothers like SPAI; and we currently investigate this approach.

Additionally, we plan to extend our implementation to the ELLPACK-T format (on top of the ELLPACK-R format which we already employ) as suggested by Vazquez et al. [24]. Obviously, we need to evaluate the performance of full 3D problems. We also expect significantly higher performance on FERMI GPUs, NVIDIA's most recent architecture. Our CPU results indicate that the ELLPACK-R format is very beneficial in terms of cache utilisation, and FERMI is the first GPU generation to include a chip-global L2 cache.

## Acknowledgements

This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under the grant TU 102/22-2, and by BMBF (call: HPC Software für skalierbare Parallelrechner) in the SKALB project (01IH08003D / SKALB). We would like to thank Michael Köster for help with FEAT2 and Jens F. Acker for the BLOOD coarse grid.

# References

[1] M. Garland, D.B. Kirk, "Understanding throughput-oriented architectures", *Commununications of the ACM*, 53(11): 58–66, Nov. 2010.

[2] J.D. Owens, M. Houston, D.P. Luebke, S. Green, J.E. Stone, J.C. Phillips, "GPU Computing", *Proceedings of the IEEE*, 96(5): 879–899, May 2008.

[3] M. Garland, S.L. Grand, J. Nickolls, J.A. Anderson, J. Hardwick, S. Morton, E.H. Phillips, Y. Zhang, V. Volkov, "Parallel Computing Experiences with CUDA", *IEEE Micro*, 28(4): 13–27, July 2008.

[4] S. Sengupta, M. Harris, M. Garland, J.D. Owens, "Efficient Parallel Scan Algorithms for many-core GPUs", in J.J. Dongarra, D.A. Bader, J. Kurzak (Editors), *Scientific Computing with Multicore and Accelerators*, Chapter 19. CRC Press, Jan. 2011.

[5] W. Hackbusch, *Multi-grid methods and applications*, Springer, Oct. 1985.

[6] M. Köster, S. Turek, "The influence of higher order FEM discretisations on multigrid convergence", *Computational Methods in Applied Mathematics*, 6(2): 221–232, June 2006.

[7] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, "Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid", *ACM Transactions on Graphics*, 22(3): 917–924, July 2003.

[8] N. Goodnight, C. Woolley, G. Lewin, D.P. Luebke, G. Humphreys, "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware", in M. Doggett, W. Heidrich, W.R. Mark, A. Schilling (Editors), *Graphics Hardware 2003*, pages 102–111, July 2003.

[9] E. Elsen, P. LeGresley, E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU", *Journal of Computational Physics*, 227(24): 10148–10161, Dec. 2008.

[10] M.J. Molemaker, J.M. Cohen, S. Patel, J. Noh, "Low viscosity flow simulations for animations", in M. Gross, D. James (Editors), *Eurographics / ACM SIGGRAPH Symposium on Computer Animation*, July 2008.

[11] J.M. Cohen, M.J. Molemaker, "A fast double precision CFD code using CUDA", in *ParCFD'2009: 21st International Conference on Parallel Computational Fluid Dynamics*, May 2009.

[12] M. Kazhdan, H. Hoppe, "Streaming multigrid for gradient-domain operations on large images", *ACM Transactions on Graphics*, 27(3): 1–10, Aug. 2008.

[13] H. Grossauer, P. Thoman, "GPU-Based Multigrid: Real-Time Performance in High Resolution Nonlinear Image Processing", in A. Gasteratos, M. Vincze, J.K. Tsotsos (Editors), *Computer Vision Systems*, Volume 5008 of *Lecture Notes in Computer Science*, pages 141–150. Springer, May 2008.

[14] Z. Feng, P. Li, "Multigrid on GPU: Tackling Power Grid Analysis on parallel SIMT platforms", in *ICCAD 2008: IEEE/ACM International Conference on Computer-Aided Design*, pages 647–654, Nov. 2008.

[15] G. Haase, M. Liebmann, C.C. Douglas, G. Plank, "A Parallel Algebraic Multigrid Solver on Graphics Processing Units", in W. Zhang, Z. Chen, C.C. Douglas, W. Tong (Editors), *High Performance Computing and Applications*, Volume 5938 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2010.

[16] D. Göddeke, R. Strzodka, "Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid", *IEEE Transactions on Parallel and Distributed Systems, Special Issue: High Performance Computing with Accelerators*, Mar. 2010.

[17] D. Göddeke, R. Strzodka, "Mixed Precision GPU-Multigrid Solvers with Strong Smoothers", in J.J. Dongarra, D.A. Bader, J. Kurzak (Editors), *Scientific Computing with Multicore and Accelerators*, Chapter 7. CRC Press, Jan. 2011.

[18] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms", in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Nov. 2007, Article No. 38.

[19] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, N. Koziris, "Understanding the performance of Sparse Matrix-Vector-Multiplication", in *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 283–292, Feb. 2008.

[20] L. Buatois, G. Caumon, B. Lévy, "Concurrent number cruncher – A GPU implementation of a general sparse linear solver", *International Journal of Parallel, Emergent and Distributed Systems*, 24(9): 205–223, June 2009.

[21] N. Bell, M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors", in *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, Nov. 2009, Article No. 18.

[22] M.M. Baskaran, R. Bordawekar, "Optimizing Sparse Matrix-Vector Multiplication on GPUs", Technical Report RC24704, IBM Research, Dec. 2008.

[23] Y.D. Kincaid DR, Oppe TC, "ITPACKV 2D Users Guide. CNA-232", `http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/`.

[24] F.M. Vazquez, G. Ortega, J.J. Fernandez, E.M. Garzon, "Improving the Performance of the Sparse Matrix Vector Product with GPUs", in *International Conference on Computer and Information Technology (CIT 2010)*, pages 1146–1151, June/July 2010.

[25] S. Williams, N. Bell, J.W. Choi, M. Garland, L. Oliker, R. Vuduc, "Sparse Matrix-Vector Multiplication on Multicore and Accelerators", in J.J. Dongarra, D.A. Bader, J. Kurzak (Editors), *Scientific Computing with Multicore and Accelerators*, Chapter 5. CRC Press, Jan. 2011.

[26] S. Turek, "On Ordering Strategies in a Multigrid Algorithm", in *Proc. 8th GAMM–Seminar*, Volume 41 of *Notes on Numerical Fluid Mechanics*, 1992.

[27] D. Braess, *Finite Elements – Theory, Fast Solvers and Applications in Solid Mechanics*, Cambridge University Press, 2nd edition, Apr. 2001.

[28] D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke, C. Gutwenger, "HONEI: A collection of libraries for numerical computations targeting multiple processor architectures", *Computer Physics Communications*, 180(12): 2534–2543, Dec. 2009.

[29] NVIDIA Corporation, "NVIDIA CUDA Programming Guide version 3.0", `http://www.nvidia.com/cuda`, 2010.

[30] S. Mallach, C. Gutwenger, "Improved Scalability By Using Hardware-Aware Thread Affinities", in R. Keller, D. Kramer, J.P. Weiß (Editors), *Facing the Multicore Challenge*, Volume 6310 of *Lecture Notes in Computer Science*, pages 29–41. Springer, Sept. 2010.

[31] S. Turek, J. Hron, M. Mádlík, M. Razzaq, H. Wobker, J.F. Acker, "Numerical simulation and benchmarking of a monolithic multigrid solver for fluid–structure interaction problems with application to hemodynamics", in H.J. Bungartz, M. Mehl, M. Schäfer (Editors), *Fluid Structure Interaction II: Modelling, Simulation, Optimization*, Volume 73 of *Lecture notes in Computational Science and Engineering*, pages 193–220. Sept. 2010.

[32] O. Bröker, M.J. Grote, "Sparse approximate inverse smoothers for geometric and algebraic multigrid", *Applied Numerical Mathematics*, 41(1): 61–80, Apr. 2002.

[33] C. Cecka, A.J. Lew, E. Darve, "Assembly of finite element methods on graphics processors", *International Journal Numerical Methods in Engineering, accepted*, 2010.

[34] D. Komatitsch, G. Erlebacher, D. Göddeke, D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster", *Journal of Computational Physics*, 229: 7692–7714, Oct. 2010.