Towards a complete FEM-based simulation toolkit on GPUs: Unstructured Grid Finite Element Geometric Multigrid solvers with strong smoothers based on Sparse Approximate Inverses

M. Geveler*, D. Ribbrock*, D. Göddeke*, P. Zajac*, S. Turek*
Corresponding author: markus.geveler@math.tu-dortmund.de

* Institute of Applied Mathematics, TU Dortmund University of Technology, Germany.

Abstract: We describe our FE-gMG solver, a finite element geometric multigrid approach for problems relying on unstructured grids. We augment our GPU- and multicore-oriented implementation technique based on cascades of sparse matrix-vector multiplication by applying strong smoothers. In particular, we employ Sparse Approximate Inverse (SPAI) and Stabilised Approximate Inverse (SAINV) techniques. We focus on presenting the numerical efficiency of our smoothers in combination with low- and high-order finite element spaces as well as the hardware efficiency of the FE-gMG. For a representative problem and computational grids in 2D and 3D, we achieve a speedup of an average of 5 on a single GPU over a multithreaded CPU code in our benchmarks. In addition, our strong smoothers can deliver a speedup of 3.5 depending on the element space, compared to simple Jacobi smoothing. This can even be enhanced to a factor of 7 when combining the usage of Approximate Inverse-based smoothers with clever sorting of the degrees of freedom. In total the FE-gMG solver can outperform a simple, (multicore-)CPU-based multigrid by a total factor of over 40.

Keywords: unstructured grids, multigrid solvers, sparse matrices, finite elements, strong smoothers, GPU computing, SPAI, SAINV

1 Introduction

Finite element methods (FEM) are a highly accurate flexible and theoretically rigorous instrument for solving partial differential equations (PDEs) that arise in many fields. Examples include high-order and non-conforming elements over arbitrarily unstructured geometries, adaptivity, a priori/a posteriori error estimation and special Pressure-Schur-Complement preconditioning in the solution of the Navier-Stokes equations [1].

A time consuming step within the solution pipeline of FEM is the linear system solver. Both numerical efficiency and hardware efficiency have to be addressed simultaneously to achieve a good total efficiency: Geometric Multigrid (gMG) solvers can treat the arising sparse linear systems in a number of iterations that is independent of the grid width. In combination with high-order Finite Elements, even superlinear convergence effects can be obtained [2]. However,

the numerical efficiency and robustness of multigrid methods strongly depends on the smoothing operator, see section 3.2.

Over the past several years, graphics processors (GPUs) have made the transition to a valuable and increasingly accepted general purpose computing resource, both on standalone workstations and in large-scale HPC installations. 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 identical tasks, and the chips' 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 [3] for technical details and a concise description of the hardware-software model of *throughput-oriented computing*.

2 Solution approach

We present and evaluate an augmented version of a previously proposed implementation technique for FE-gMG (Finite Element Geometric Multigrid) solvers for PDE problems discretised on *unstructured* grids. Our target architectures are fine-grained (manycore) GPUs – at this point, we focus entirely on the solver performance, evaluating it for different finite element spaces. Since the smoother is the most critical part concerning performance and robustness of the gMG, we evaluate numerically efficient smoothers based on Approximate Inverses (SPAI and SAINV), see section 3.2.

In our approach, performance-critical components of the solver pipeline, the smoother, grid transfer, coarse-grid solver and defect operators, are entirely based on cascades of sparse matrix-vector multiplications (SpMV). This consequent reduction to one performance-critical kernel within the multigrid solver 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 with one central, well-understood and well-optimised parallel kernel (see section 3.1), which is favourable in terms of software maintainability, performance-tuning and the adoption of GPUs in multigrid and finite element codes.

2.1 Related work

On GPUs, multigrid methods have received only 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. [4, 5]. They used geometric and algebraic multigrid (aMG) for finite-difference type discretisations. More recent publications presenting applications that require multigrid solvers are supersonic flows (aMG, unstructured grids [6]), (interactive) flow simulations for feature film (aMG/gMG, structured [7, 8]), out-of core multigrid for gigapixel image stitching (gMG/aMG, structured [9]), image denoising and optical flow (gMG/aMG, structured [10]), power grid analysis (aMG, structured/unstructured [11]) and electric potential in the human heart (aMG, unstructured [12]). This last paper is 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 specifically designed, problem-specific data layout in their SpMV implementation whereas we go further and

use a layout that has been shown to deliver superior performance for a wide range of non-zero patterns. Also very closely related is the work by Heuveline et al., who pursue many different fine-grained parallel preconditioning techniques based on multicolouring: Simultaneously to our work, they have started to evaluate their preconditioners as smoothers in the multigrid context [13]. 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, together with Heuveline and his co-workers we are the first to present *geometric MG with strong smoothers for high-order unstructured grid FEM on GPUs*.

2.2 Contribution and paper outline

The paper at hand is the second in a series concerning FE-gMG on GPUs, significantly expanding a previous conference proceedings [14] and the initial publication on the topic which did not address strong smoothers [15]. The FE-gMG in this paper is based on a better matrix storage format (ELLPACK-T, see section 3.1), and we focus on the evaluation of different preconditioners (Jacobi, SPAI and SAINV) in combination with different conforming finite element spaces (Q_1 and Q_2). Finally, all results are calculated on a newer generation of hardware (especially Fermi-type GPUs), see section 4.

Throughout our computations, we solely concentrate on the solution of the linear system, which means, that all needed matrices (e.g. stiffness- and transfer matrices as well as the preconditioners given by sparse Approximate Inverses within the smoother) are preassembled; this topic is subject to another publication under preparation. This is justified since in many practical scenarios, the linear solver dominates the total execution time and the combined effects of advanced smoothers, suitable numbering of the degrees of freedom and feasible utilisation of hardware acceleration have to be analysed. Here, the former increases numerical efficiency while simultaneously increasing the complexity of the SpMV operation by increasing the number of non-zeros and altering the sparsity pattern; the latter two are strongly related to each other since the sorting strategy directly influences matrix-bandwidth which is a major criterion for the performance of the GPU SpMV. In our publications mentioned above, we have already analysed the impact of the numbering of the degrees of freedom especially when using the GPU. Hence, in this paper, we concentrate on the other aspects mentioned above. We continue by describing the components of the FE-gMG solver in detail in section 3, where we focus on the smoother and grid transfer operators. Especially, section 3.1 is dedicated to the SpMV kernel and the implementational aspects of FE-gMG. In section 4 we present results for our approach applied to a common model problem. Finally, we give a concise conclusion in section 5.

3 FE-gMG – Finite Element Geometric Multigrid

3.1 SpMV kernel

We do not utilise the 'standard' CSR format, but rather the ELLPACK-T format proposed and exemplarily implemented by Vazques et al. [16]. In our experience, ELLPACK(-T) leads to significantly higher computational throughput, independent of the architecture and even for sequential code.

With the ELLPACK-T format, the sparse matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$ is performed by computing each entry y_i of the result vector $\mathbf{y}$ independently. In general, this results in a 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, and due to the indirect addressing, memory access can be arbitrarily scattered.

The ELLPACK-T based SpMV kernel is mapped to the GPU architecture by launching one or more device threads for the calculation of an entry y_i, resulting in fully coalesced memory access to the matrix and the vector $\mathbf{y}$ 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, the device-wide L2-cache is well utilised. No synchronisation between threads is necessary. The threads in one CUDA warp do not diverge because flow instructions are not necessary which would cause serialisation. Every warp finishes execution directly when all non-zero entries in the rows of its threads are completely processed. Because of this, only warps with a high relative non-zero count within their rows execute longer compared to average warps.

Finally, the ELLPACK-T format augments the former ELLPACK-R by storing t elements of one row contiguously in memory and thus allows for multiple threads in one warp to process one row.

3.2 Smoothing operator

As a key component for multigrid, our *smoother* is realised as a damped preconditioned Richardson iteration, using SpMV to plug in the preconditioner and for defect calculation:

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \omega M(\mathbf{b} - A\mathbf{x}^k)$$

Here, $A\mathbf{x} = \mathbf{b}$ is the linear system to solve and $M \approx A^{-1}$ a preassembled (sparse) preconditioner that approximates the inverse of the stiffness matrix. In this paper, we use three different approaches to construct our smoother:

- Jacobi is used as a representative of a simple 'standard' smoother: $M_{\mathrm{JAC}} = \mathrm{diag}(A)^{-1}$.
- In order to gain a better approximation of the inverse, the SPAI (*Sparse Approximate Inverse*) technique relies on the fact that one can minimise

$$\| I - MA \|_F^2 \quad = \quad \sum_{k=1}^{n} \| \mathbf{e}_k^{\mathrm{T}} - \mathbf{m}_k^{\mathrm{T}} A \|_2^2 \quad = \quad \sum_{k=1}^{n} \| A^{\mathrm{T}}\mathbf{m}_k - \mathbf{e}_k \|_2^2$$

  where $\mathbf{e}_k$ is the k-th unit-vector and $\mathbf{m}_k$ is the k-th column of M. Therefore it follows that for n columns of M we solve n independent and small *least-squares* optimisation problems to construct $M_{\mathrm{SPAI}} = [\mathbf{m}_1, \mathbf{m}_2, \ldots \mathbf{m}_k]$:

$$\min_{m_k} \| A^{\mathrm{T}}\mathbf{m}_k - \mathbf{e}_k \|_2, \;\; k = 1, \ldots n, .$$

  The SPAI algorithm has been initially proposed by Grote and Huckle [17], and similar to ILU-type techniques, an entire family of preconditioners is obtained depending on the level of fill-in allowed during the construction process. Typical variants of the SPAI procedure restrict the fill-in to the main diagonal (SPAI(0)) or to the non-zero pattern of the stiffness matrix (SPAI(1)). It has been reported that SPAI(0) has approximately the same smoothing properties as damped Jacobi, while SPAI(1) can be compared to Gauß-Seidel [18]. In this paper, we use the latter SPAI algorithm in our benchmarks.

- An algorithm to compute an even better sparse preconditioner is SAINV (*Stabilised Approximate Inverse*, see Benzi et al. [19]), which is based on the calculation of a factorisation

$A^{-1} = ZD^{-1}Z^T$ where Z and D can be explicitly calculated via an A-biconjugation process applied to the unit basis vectors. Z is computed incompletely, by removing elements less than a prescribed drop tolerance, to gain the incomplete factorisation M_{SAINV}. As opposed to SPAI, it is not possible to predefine the resulting sparsity pattern of M_{SAINV}. The dynamic distribution of non-zeros offers the advantage that the resulting pattern may match the actual inverse of the stiffness matrix much better than a predefined one. The properties of SAINV as a preconditioner in a linear system solver have already been analysed [20, 21], and the numerical capabilities of SAINV roughly correspond to those of ILU(0). However, SAINV as a smoother within a gMG has not received much attention yet.

Note, that in all three cases, we precompute a single Approximate Inverse M_{JAC}, M_{SPAI} or M_{SAINV} respectively which is plugged into the base iteration via a single SpMV (or element-wise vector-vector product in case of the Jacobi smoother). In the case of SAINV, this optimisation is possible as we only consider symmetric stiffness matrices, so that the sparse matrix matrix multiplication does not change the non-zero pattern.

At this point, we concentrate our efforts solely on the solution process. However, it should be noted here, that for both Approximate Inverse techniques the original publications state, that setup costs for the corresponding matrices are moderate. We therefore believe that they can be precomputed efficiently and alleviate this topic to future work.

3.3 Grid transfer operators

Furthermore, *grid transfer* can be implemented by means of SpMV, if we choose the standard Lagrange bases for two consecutively refined Q_k finite element spaces V_{2h} and V_h. In this case, we can interpolate any function $u_{2h} \in V_{2h}$ in order to prolongate it to V_h and the interpolant u_h can be calculated by evaluating u_{2h} in the corresponding nodal points $\xi_h^{(i)}$ of the basis function $\varphi_h^{(i)}$:

$$u_h := \sum_{i=1}^{m} x_i \varphi_h^{(i)}, \quad x_i := u_{2h}(\xi_h^{(i)})$$

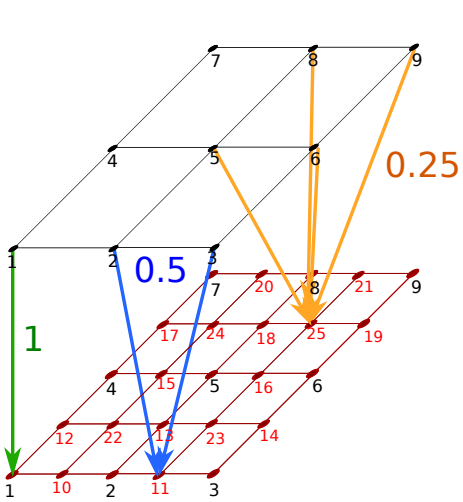
For the basis functions of V_{2h} and $u_{2h} = \sum_{j=1}^{n} y_j \varphi_{2h}^{(j)}$ with coefficient vector y, we can write the prolongation analogously as

$$u_h := \sum_{i=1}^{m} x_i \varphi_h^{(i)}, \quad x := P_{2h}^h \cdot y$$

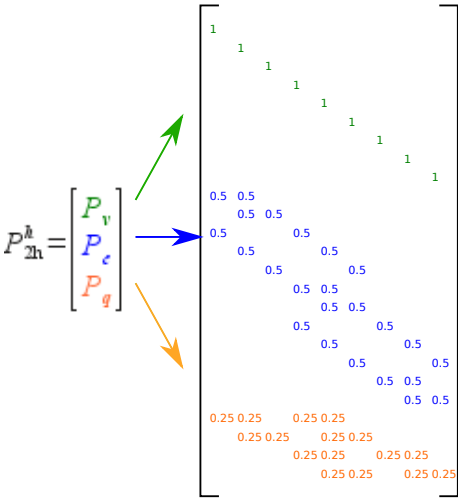
which results in an $m \times n$ *prolongation matrix* $(P_{2h}^h)_{ij} = \varphi_{2h}^{(j)}(\xi_h^{(i)})$ which can be transposed to retrieve the corresponding restriction matrix. Figures 1(a) and (b) depict a basic (structured) example for a 2D quadrilateral grid and a two-level numbering scheme. The storage demand of the prolongation matrix is bounded by the memory requirements of the discrete Laplace operator on V_h and the bandwidth strongly depends on the numbering technique used to label the degrees of freedom (see section 4), which of course also holds true for the associated system matrices.

3.4 Coarse grid solver

Finally, Krylov-subspace solvers are employed as coarse grid solver which are also composed solely of SpMV and vector-vector operations. Note that applying a non-symmetric precondi-



(a) 2D Q_1 interpolation: Each vertex of the fine grid corresponds to either a vertex, an edge midpoint or a quadrilateral midpoint in the coarse grid



(b) resulting block prolongation matrix for two-level numbering: Node vertices of the coarse grid need no interpolation resulting in an identity matrix block; coarse edges' corner vertices contribute with weight 1/2 to interpolation for coarse edge midpoints; corner vertices of coarse quadrilaterals are represented with weight 1/4 in the prolongation matrix

Figure 1: Construction of prolongation matrices.

tioner requires a feasible method here like BiCGStab, which we use in all our benchmarks.

With these three major components (smoother, transfer and coarse grid solver) being based on the matrix-vector multiply, only some vector-vector numerical linear algebra remains to be implemented alongside a defect-operator which is also only little more than a regular SpMV.

4 Benchmark setup and performance results

4.1 Problem- and solver setup

As a representative of an elliptic PDE we employ the Poisson problem, well-known as a fundamental building block (and often the computational bottleneck) in computational fluid dynamics (pressure-Poisson problems in operator-splitting approaches) and other areas. The domain has two boundary components, one rectangular outer boundary component Γ_1 and one for the circular inner boundary (Γ_2), both equipped with Dirichlet boundary conditions; analogously in 3D:

$$\begin{cases} -\Delta u = 1, & \mathbf{x} \in \Omega \\ u = 0, & \mathbf{x} \in \Gamma_1 \\ u = 1, & \mathbf{x} \in \Gamma_2 \end{cases}$$

The domain is then covered by an unstructured grid, see Figure 2 for the 2D benchmark and Figure 3 for the 3D case. The fine grid resolution is configured to maintain equal numbers of degrees of freedom of approximately 0.5×10^6 (2D) for both finite elements, resulting in a non-zero count (in the stiffness matrices) of around 4.8×10^6 for the quadrilateral Q1 element

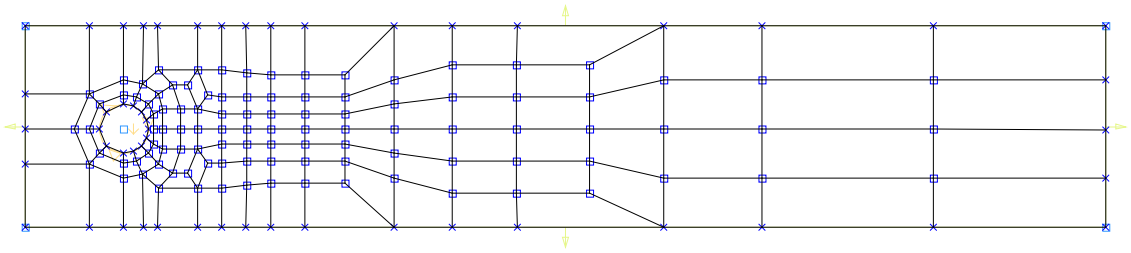


Figure 2: Grid of the 2D benchmark problem ($\square \Rightarrow \mathbf{x} \in \Omega$, $\times \Rightarrow \mathbf{x} \in \Gamma_1 \cup \Gamma_2$).

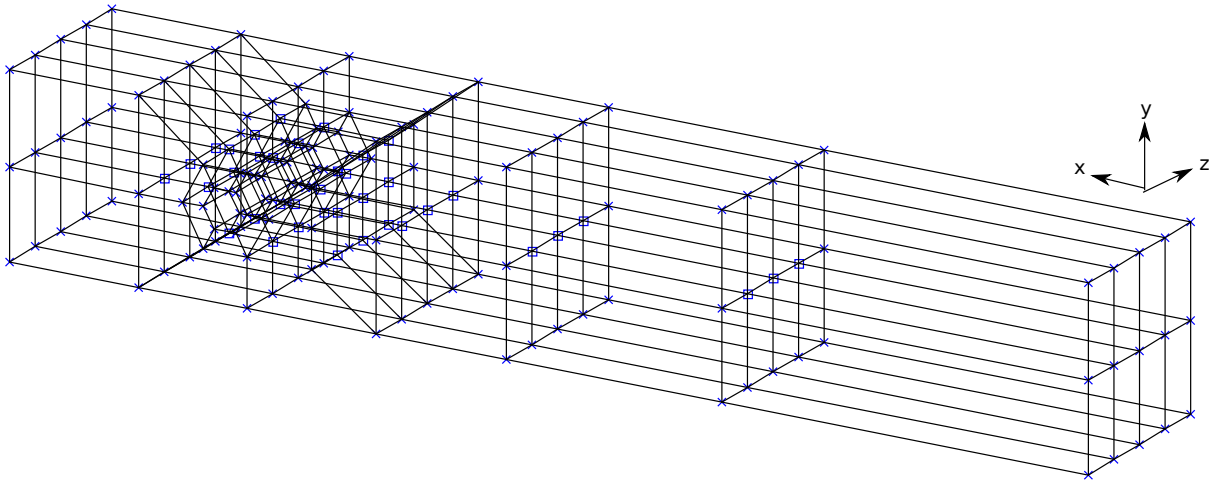


Figure 3: Grid of the 3D benchmark problem ($\square \Rightarrow \mathbf{x} \in \Omega$, $\times \Rightarrow \mathbf{x} \in \Gamma_1 \cup \Gamma_2$).

and 8.5×10^6 for Q2 respectively. In 3D, the maximum number of degrees of freedom in both, Q1 and Q2 is approximately 0.1×10^6 comprising numbers of non-zero elements of around 1.3×10^6 for Q1 and 2.8×10^6 for Q2 respectively. We configure our multigrid solver for traversing a full mesh hierarchy using an F-Cycle and eight pre- and postsmoothing steps (2D) and both 16 pre- and postsmoothing steps in the 3D case respectively. In particular, we apply 7/6 multigrid levels for Q1/Q2 in case of the 2D benchmark. The multigrid hierarchy in 3D is 4/3 levels deep for Q1/Q2.

Coarse grid problems are treated with a BiCGStab solver which is configured to reduce the initial residual by two digits. In addition, we use three increasingly strong smoothers described in section 3.2, namely the Richardson iteration with preconditioners M_{JAC}, M_{SPAI} and M_{SAINV} respectively. In our experiments, we use a damping of $\omega = 0.5$ for the Jacobi and SAINV smoothers and $\omega = 1$ when using SPAI. We examined the SAINV drop tolerance $\epsilon = 0.013$ to deliver a good tradeoff between the number of non zero entries of M_{SAINV} and its smoothing properties. Note, that for the Approximate Inverses, numbers of non-zero elements may vary by the sorting strategy for the degrees of freedom (see below).

All benchmarks are performed on an Intel Core i7 980 hexacore (Gulftown) workstation (memory bandwidth: 35 GB/s) including an NVIDIA Tesla C2070 GPU (with around 140 GB/s) and we use different techniques for numbering the degrees of freedom: Two-level ordering (**2LV**), Cuthill McKee ordering (**CM**), coordinate-based ordering (**XYZ**) pseudo-random ordering (**Stoch**) and hierarchical ordering (**Hie**) (see Turek et al. for details on these approaches [22]). All benchmarks are carried out in double precision.

Note that we use a Pthreads-driven MultiCore framework in our CPU benchmarks utilising an architecture-optimal number of six threads (see Geveler et al. for implementational details [23]).

A reduction of the initial residual by eight digits is employed as stopping criterion:

$$\|\mathbf{b} - A\mathbf{x}^n\|_2 / \|\mathbf{b} - A\mathbf{x}^0\|_2 \leq \epsilon := 10^{-8} \tag{1}$$

We emphasise, that all results are numerically equal to those computed by a serial reference multigrid solver.

4.2 Performance measurements

The execution times, numbers of iterations and speedups for quadrilateral Q_1 and Q_2 elements are displayed in figures 4 for the 2D benchmark and 5 for the 3D case respectively. When taking a look at the benchmark data, we can consolidate our findings from our previous work: A speedup factor of 5 on average can be achieved when using the GPU instead of the (multi-threaded) CPU solver, which is in perfect accordance with the memory-bandwidth difference between these two architectures.

In addition, clever sorting (that reduces matrix bandwidth, e.g. the **XYZ** technique) makes a difference: Independent of the applied smoother, we can achieve a speedup of up to two over arbitrarily numbered degrees of freedom (emulated when using the **Stoch** sorting).

Q1	CPU								GPU											
	Jacobi		SPAI			SAINV			Jacobi			SPAI				SAINV				
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	4.04	13	2.54	5	1.59	3.59	6	1.12	1.06	13	3.82	0.56	5	1.88	4.53	1.19	6	0.89	3.01
CM	3.65	13	2.19	5	1.66	3.29	6	1.11	1.03	13	3.55	0.72	5	1.43	3.05	0.82	6	1.26	4.03
XYZ	3.48	13	2.06	5	1.69	4.44	9	0.78	0.98	13	3.53	0.51	5	1.93	4.04	1.03	9	0.96	4.32
Stoch	4.04	13	2.57	5	1.57	3.19	5	1.27	1.74	13	2.33	1.04	5	1.66	2.46	1.29	5	1.35	2.47
Hie	3.49	13	2.07	5	1.69	3.07	6	1.14	0.97	13	3.59	0.50	5	1.94	4.14	0.77	6	1.26	3.98

Q2	CPU								GPU											
	Jacobi		SPAI			SAINV			Jacobi			SPAI				SAINV				
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	13.19	22	4.87	5	2.71	10.24	9	1.29	2.27	22	5.80	0.93	5	2.44	5.22	2.04	9	1.12	5.02
CM	11.40	22	4.40	5	2.59	10.58	12	1.08	2.50	22	4.56	1.02	5	2.44	4.30	2.20	12	1.13	4.80
XYZ	11.29	22	4.21	5	2.68	9.58	12	1.18	2.41	22	4.69	0.99	5	2.44	4.26	2.70	12	0.89	3.55
Stoch	12.92	22	5.14	5	2.51	4.64	9	2.79	4.78	22	2.70	2.04	5	2.35	2.52	1.57	9	3.05	2.96
Hie	11.25	22	4.24	5	2.66	8.68	9	1.30	2.44	22	4.60	1.00	5	2.43	4.22	1.88	9	1.30	4.61

Figure 4: Total execution times of the FE-gMG (2D case).

Especially using our sophisticated SPAI-based smoother allows for the solution process to be numerically more efficient: The numbers of needed iterations are reduced by more than a factor of two to four compared to the Jacobi procedure in our 2D benchmark. In 3D, this factor can even reach three to five dependent of the finite element space. This means, that in all cases, the SPAI-based smoother speeds up the multigrid procedure by at least 50 percent and that it can even reduce the overall solution time by a factor of 3.5. In addition, the application of

Q1	CPU								GPU							
	Jacobi		SPAI			SAINV			Jacobi			SPAI			SAINV			
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	2.43	26	1.08	7	2.25	1.03	9	2.37	0.66	26	3.71	0.27	7	2.39	3.94	0.28	9	2.32	3.63
CM	2.34	26	1.02	7	2.30	0.98	9	2.37	0.66	26	3.53	0.28	7	2.39	3.67	0.29	9	2.26	3.36
Stoch	2.63	26	1.18	7	2.23	1.28	10	2.06	0.75	26	3.48	0.33	7	2.32	3.61	0.38	10	1.98	3.35

Q2	CPU								GPU							
	Jacobi		SPAI			SAINV			Jacobi			SPAI			SAINV			
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	9.86	42	3.09	8	3.19	2.44	10	4.04	2.01	42	4.90	0.58	8	3.44	5.29	0.56	10	3.60	4.37
CM	7.46	42	2.50	8	2.99	2.41	12	3.10	2.31	42	3.23	0.70	8	3.32	3.59	0.73	12	3.18	3.32
Stoch	8.89	42	3.14	8	2.83	2.90	12	3.07	2.92	42	3.04	0.92	8	3.18	3.41	0.92	12	3.19	3.16

Figure 5: Total execution times of the FE-gMG (3D case).

SPAI to the smoothing step is very robust: It does not require any damping and is invariant to permuting the matrix (as it is done by resorting the degrees of freedom).

When taking a look at the results for SAINV, it can be found, that the smoother reduces the numbers of iterations by a factor of two to four compared to Jacobi and in most cases the computational costs can be reduced varying from a speedup of some percent up to three compared to simple Jacobi smoothing. This is regardless the fact that a single SpMV operation with a SAINV matrix is somewhat more complex (due to a denser sparsity pattern) than an SpMV applied to a SPAI matrix. However, in some few cases the application of the SAINV-based smoother does not pay off. The reason for this is that since the damping parameter ω is defined homogeneously (which is the only realistic and reasonable approach) among all benchmarks and preliminary experiments showed that SAINV reacts sensibly on damping, not all measurements presented here can be considered to be optimal. This leads to worse (as compared to SPAI) convergence rates and the gain (as compared to Jacobi) in convergence rate and the additional computational cost due to the more complex SpMV operation may be disproportional.

However in the 3D benchmark, application of the SAINV-based smoother reaches the performance of SPAI and on the CPU, it is even sometimes faster. Here, the associated SAINV Approximate Inverses contain somewhat less non-zero elements than the SPAI matrices. This is due to the abovementioned dropping criterion used in the creation of the non-zero layout of the SAINV preconditioner. In addition, the complexity of the SpMV in the smoothing operator seems to be coped with slightly better by the CPU due to the fact, that the matrix-vector product involves some data-reuse and the CPU can exploit its more advanced cache-hierarchy.

5 Conclusion and future work

Our flexible and efficient, hardware-oriented FE-gMG for unstructured grids, based on cascades of SpMV, has been successfully augmented: On the one hand, we have enhanced the performance of the SpMV kernel for the GPU by switching to the currently best-fitting data storage format for sparse matrices: ELLPACK-T. In addition, our way of setting up smoother operators out of a base iteration with a preconditioner given by a sparse Approximate Inverse allows for the application of strong smoothing for problems involving unstructured grids. We have begun to demonstrate, that even for quite simple problems, the more complex SpMV operation in all smoothing steps induced by denser Approximate Inverses may return a good profit when using the FE-gMG: Total execution times may be reduced by a factor of around 3.5 due to increased convergence-rates in addition to possible speedup of up to six when using the GPU. In combination with clever sorting of the degree of freedom, we can generate an overall performance

gain in the FE-gMG that reaches a factor of more than 40 compared to our baseline multicore CPU multigrid solver with Jacobi smoothing. Note, that GPU acceleration on its own yields the largest fraction of this speedup. However the combined effects of better sorting and tailored smoothing together even exceeds this factor, which encourages us in further pursueing our aim of enhancing numerical- and hardware-efficiency simultaneously (*Hardware-oriented Numerics*).

Our work so far has opened the door to a large space of experiments: The numerical (cross-) effects of preconditioners in the smoother, damping, finite element space, cycle-type (V- W- or F-types and nested types) and sorting of the degrees of freedom in the computational mesh have to be analysed carefully. Ultimately the assembly of stiffness- and matrices approximating their inverses as well as transfer matrices yields new challenges: Here, storage formats allowing for fast or even random access are needed, which of course have to be compatible with those used in the solvers. The work by Cecka et al. can serve as a good starting point for this avenue [24].

Therefore, we plan to continue this series with work addressing both, numerically efficient FE-gMG variants and hardware-oriented kernels needed by it.

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). Thanks to the Dortmund ITMC team for access to the LiDO cluster.

References

[1] S. Turek, *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*, Springer, Berlin, 1999, ISBN 3-540-65433-X.

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

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

[4] 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, 2003.

[5] 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, 2003.

[6] 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, 2008.

[7] 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*, 2008.

[8] 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*, 2009.

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

[10] 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, 2008.

[11] 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, 2008.

[12] 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.

[13] V. Heuveline, D. Lukarski, N. Trost, J.P. Weiss, "Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs", Technical Report ISSN: 2191-0693, Institut für Angewandte und Numerische Mathematik (Inst. f. Angew. u. Num. Math.), Karlsruhe Institute of Technology, 2011.

[14] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, S. Turek, "Towards a complete FEM-based simulation toolkit on GPUs: Geometric Multigrid solvers", in *23rd International Conference on Parallel Computational Fluid Dynamics (ParCFD'11)*, 2011.

[15] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, S. Turek, "Efficient Finite Element Geometric Multigrid Solvers for Unstructured Grids on GPUs", in P. Iványi, B.H. Topping (Editors), *Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, page 22, 2011.

[16] 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, 2010.

[17] M.J. Grote, T. Huckle, "Parallel Preconditioning with Sparse Approximate Inverses", *SIAM Journal on Scientific Computing*, 18: 838–853, 1996.

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

[19] M. Benzi, J.K. Cullum, M. Tuma, "Robust approximate inverse preconditioning for the conjugate gradient method", *SIAM Journal on Scientific Computing*, 22(4): 1318–1332, 2000.

[20] M. Benzi, M. Tuma, "Numerical Experiments With Two Approximate Inverse Preconditioners", *BIT*, 38: 234–241, 1998.

[21] M. Tuma, M. Benzi, M. Benzi, "A Comparative Study Of Sparse Approximate Inverse Preconditioners", *Applied Numerical Mathematics*, 30: 305–340, 1998.

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

[23] M. Geveler, D. Ribbrock, S. Mallach, D. Göddeke, S. Turek, "A Simulation Suite for Lattice-Boltzmann based Real-Time CFD Applications Exploiting Multi-Level Parallelism on modern Multi- and Many-Core Architectures", *Journal of Computational Science*, 2: 113–123, 2011.

[24] C. Cecka, A.J. Lew, E. Darve, "Assembly of finite element methods on graphics processors", *International Journal for Numerical Methods in Engineering*, 85(5): 640–669, 2011.