Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid

Dominik Göddeke and Robert Strzodka

Abstract—We have previously suggested mixed precision iterative solvers specifically tailored to the iterative solution of sparse linear equation systems as they typically arise in the finite element discretization of partial differential equations. These schemes have been evaluated for a number of hardware platforms, in particular single precision GPUs as accelerators to the general purpose CPU. This paper reevaluates the situation with new mixed precision solvers that run entirely on the GPU: We demonstrate that mixed precision schemes constitute a significant performance gain over native double precision. Moreover, we present a new implementation of cyclic reduction for the parallel solution of tridiagonal systems and employ this scheme as a line relaxation smoother in our GPU-based multigrid solver. With an alternating direction implicit variant of this advanced smoother we can extend the applicability of the GPU multigrid solvers to very ill-conditioned systems arising from the discretization on anisotropic meshes, that previously had to be solved on the CPU. The resulting mixed precision schemes are always faster than double precision alone, and outperform tuned CPU solvers consistently by almost an order of magnitude.

Index Terms—GPU Computing, Mixed Precision Iterative Refinement, Multigrid, Tridiagonal Solvers, Cyclic Reduction, Finite Elements, NVIDIA CUDA.

1 Introduction

1.1 Background

Geometric multigrid solvers are the most efficient methods for the solution of finely discretized elliptic partial differential equations (PDEs). Best results are obtained when the multigrid cycle type, the number of pre- and postsmoothing steps and the smoother type are adapted to the problem. Black-box multigrid solvers are variants that utilize very strong smoothers and typically W-cycles to ensure convergence with the same parameter settings for large problem classes. In general, the more ill-conditioned a system is, the stronger the smoother has to be in the multigrid cycle.

Because of their favorable numerical properties there is a great interest in parallel implementations of multigrid solvers. Extensive work has been performed in this direction for SMP systems and clusters. When parallelizing across multiple processors, the communication between them quickly becomes the bottleneck and other methods, e.g., domain decomposition, have to be used or combined with them.

When multiple cores reside on the same chip the communication between them is not dominating anymore and other performance characteristics come into play. In particular, on graphics processors (GPUs) as the first representatives of many-core devices, multiple factors must be taken into account: Performance is mainly influenced by the ability of implementations to benefit from fine-grained per-block synchronization via small but fast shared memories, and by hiding latencies of off-chip memory accesses via many active blocks on each core, realized through massive hardware-multithreading. The most prominent example of a current many-core architecture is NVIDIA CUDA [1], and we refer to several excellent survey articles for more details on GPU Computing with CUDA [2], [3] and the underlying hardware architecture and programming model [4], [5].

The transfer operations between grids parallelize naturally as do basic smoothers like the Jacobi method or a red-black SOR solver. Therefore, multigrid solvers with the Jacobi smoother and a simple V-cycle have been demonstrated early on the GPU using 'legacy GPGPU' techniques based on graphics APIs [6], [7], [8]. Earlier GPUs did not support double precision and the above references discuss only single precision multigrid schemes. A mixed precision iterative refinement scheme allows us to recover double precision results through additional updates on the CPU in this case [9]. In this paper we can perform these double precision updates on the GPU itself, enabling higher performance and broadening the applicability of GPU-based techniques to the case where single precision alone fails. More recent publications reporting on multigrid solvers with simple smoothers in CUDA include stitching of gigapixel images [10], power grid analysis [11] and the solution of the Euler equations [12].

More advanced smoothers are more difficult to handle in parallel. Kass et al. [13] presented the first GPU implemention of the alternating direction implicit tridiagonal

- *D. Göddeke is with the Department of Applied Mathematics, TU Dortmund, Germany.*
  E-mail: dominik.goeddeke@math.tu-dortmund.de
- *R. Strzodka is with the Max Planck Institut Informatik, Saarbrücken, Germany.*
  E-mail: strzodka@mpi-inf.mpg.de

This work has been partly supported by Deutsche Forschungsgemeinschaft, projects TU102/22-1 and TU102/22-2; and by Bundesministerium für Bildung und Forschung, project 01IH08003D/SKALB.

solver (ADI-TRIDI) based on shading languages. Later Sengupta et al. [14] used on-chip memory with CUDA to obtain faster results. Both parallelization approaches are based on classical cyclic reduction [15]. Two other parallel algorithms for the solution of tridiagonal equation systems are parallel cyclic reduction [16] and recursive doubling [17]. Recently, Zhang et al. [18] discussed the applicability of these algorithms on modern GPUs. They conclude that cyclic reduction suffers from shared memory bank conflicts (see Section 3) and poor thread utilization in lower stages of the solution process, while parallel cyclic reduction is not asymptotically optimal and recursive doubling is not optimal and additionally exhibits numerical stability issues. They propose a hybrid combination of cyclic reduction and parallel cyclic reduction to alleviate these deficiencies.

1.2 Paper Contribution

Simultaneously to the work by Zhang et al. [18], we have developed a cyclic reduction implementation that does not exhibit the above mentioned memory access problems at the expense of 50% more on-chip storage. Our variant significantly outperforms a standard cyclic reduction implementation, and reaches the same performance as their best hybrid algorithm.

Using this solver in an alternating direction implicit (ADI) iteration as a multigrid smoother, we are the first to assemble a multigrid solver capable of dealing with very anisotropic meshes and therefore severely ill-conditioned systems entirely on the GPU. Finally, we almost double its performance by executing our previous mixed precision iterative refinement scheme now completely on the GPU.

1.3 Paper Organization

Section 2 motivates the use of mixed precision methods and explains how our schemes can be applied in a bigger framework. Section 3 discusses different multigrid smoothers and presents our fast implementation of the cyclic reduction. In Section 4 we lay out our test cases and solver configurations and Section 5 discusses the accuracy and performance results. We conclude in Section 6.

2 Why Mixed Precision?

2.1 Test Problem

We solve the Poisson problem $-\Delta \mathbf{u} = \mathbf{f}$ on isotropic and anisotropic domains Ω with homogeneous Dirichlet boundary conditions. The Poisson problem is a very important prototypical representative of the class of elliptic PDEs, and it often appears as a sub-problem in simulation codes: Examples include the Pressure-Poisson problem when solving the Navier-Stokes equations in fluid dynamics with an operator-splitting approach, linearized elasticity (displacements in compressible materials subject to small deformations under external load) or electrostatics (potential calculations). Conforming bilinear quadrilateral elements are used for the spatial finite element discretization of rectangular domains $\Omega = [0, a] \times [0, b]$. We use a conforming (yet possibly anisotropic) subdivision scheme, so that the resulting mesh consists of $N = (2^L + 1)^2$ mesh points for a *refinement level* of $L = 1, \ldots, 10$. We obtain a linear system of equations

$$A_L \mathbf{x} = \mathbf{b},$$

where A_L is an $N \times N$ sparse matrix with exactly nine nonzero bands at known offset positions, because of the regular structure of the mesh. This *logical tensor product structure* is independent of the location of the mesh points, and is preserved even for extremely deformed meshes occurring for instance in the context of r-adaptivity [19]. The performance of low-level linear algebra routines consequently depends only on the special matrix structure and not on the underlying mesh.

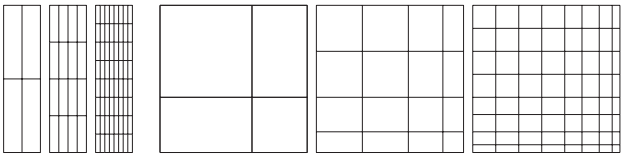

Fig. 1. Examples of different mesh anisotropies on levels 1–3. Left: Uniform subdivision of an anisotropic coarse mesh. Right: Anisotropic subdivision of the rightmost and bottommost element layer in each refinement step.

In our setting, a mixed precision solver *must* deliver the same accuracy as if computing entirely in high precision, in particular in case of strongly varying coefficients and thus high condition numbers. In our previous paper [9] we utilized test matrices arising from anisotropic mesh refinements, which is a common case in finite element based simulations. In this paper, we continue to evaluate the same test problems:

Uniform anisotropies: Using rectangular instead of square elements allows us to specify the aspect ratio of each element in the refined discretization. We refine the mesh uniformly (cf. Figure 1 on the left) which leads to the same degree of anisotropy in each entry of the matrix.

Anisotropic refinement: In this strategy, we start with the unit square and in each refinement step, we subdivide the bottommost and rightmost layer of elements anisotropically (cf. Figure 1 on the right). This refinement scheme is often used to accurately resolve boundary layers in fluid dynamics, or discontinuities in solid mechanics. For each refined element in these layers, the new midpoint x_c is calculated by recursively applying the formula $x_c = x_l + \nu \cdot \frac{x_r - x_l}{2}$ with a given *anisotropy factor* ν ($\nu = 1.0$ yields uniform refinement) and x_l and x_r denoting the x-coordinates of the left and right edge of an element before subdividing (analogously

for the y-component). All other elements are refined uniformly. This leads to matrices with locally condensed anisotropies instead of uniform ones and thus higher condition numbers.

We note that the above cases generate matrices which also occur when discretizing anisotropic operators like $\mathrm{div}(\mathbf{G}\nabla\mathbf{u})$. For example, if $\mathbf{G}$ is a constant diagonal matrix then the uniform anisotropic refinement leads to the same stiffness matrix, and similarly for anisotropic refinement and spatially varying coefficients of the matrix $\mathbf{G}$. Consequently, our test results are also relevant for problems with operator anisotropy.

2.2 The Bigger Picture

The test configurations in this paper represent an important building block in our finite element based discretization and solver toolkit FEAST [20], [21], [22]. FEAST is designed to combine high performance computing techniques with state-of-the-art numerical methodology, and executes on large-scale distributed memory machines. The basic idea in FEAST is to cover the computational domain with an unstructured collection of subdomains, each of which has the logical tensor product structure. FEAST thus balances high performance (locality, no indirect memory accesses) of the local components with the flexibility to resolve arbitrary domains in an unstructured way. We have previously reported on the solution of large-scale problems in solid mechanics and fluid dynamics on GPU-accelerated clusters [23], [24] and suggested a *minimally invasive* approach to include accelerator hardware like GPUs into a matured MPI-based package, we refer to these publications for details.

All previous results employed a GPU-accelerated multigrid with simple Jacobi smoothing only, advanced smoothers had only been available on the CPU at that time. In this paper, we aim at closing this gap by evaluating the performance of an advanced smoother, specifically tailored to the underlying logical tensor product structure, for the prototypical 'one subdomain' configuration. In view of the bigger picture, the entire MPI infrastructure, the domain decomposition and the parallel multilevel solver employed by FEAST benefit directly from the improvements suggested in this paper, as the entire solution scheme is in fact a mixed precision scheme already.

2.3 Single Precision Accuracy

To measure accuracy, we evaluate the analytical Laplacian of the polynomial function $u_0(x,y) = x(a-x)y(b-y)$ at the grid points and use the resulting coefficient vector as the right hand side of the linear system; a and b are chosen so that the function always vanishes on the boundary of the rectangular test domains. We thus know the exact solution of the problem to be solved, and can calculate the error in the l_2 norm by integrating over the domain. According to finite element theory, the error reduces by a factor of four (h^2 for the mesh width h) after refining each element into four smaller ones.

TABLE 1
Influence of solver precision on the accuracy of the solution with increasing level of refinement (L) and total problem size (N).

		single precision		double precision	
L	N	Error	Red.	Error	Red.
2	25	7.1663649E-2		7.1663606E-2	
3	81	1.7802522E-2	4.03	1.7802586E-2	4.03
4	289	4.4428836E-3	4.01	4.4429161E-3	4.01
5	1,089	1.1103626E-3	4.00	1.1102363E-3	4.00
6	4,225	2.7897810E-4	3.98	2.7752805E-4	4.00
7	16,641	5.5209742E-5	5.05	6.9380191E-5	4.00
8	66,049	2.8037403E-5	1.97	1.7344895E-5	4.00
9	263,169	2.2419906E-4	0.13	4.3362264E-6	4.00
10	1,050,625	1.0585913E-3	0.21	1.0841185E-6	4.00

Even for the fully isotropic case (U1 configuration, see Section 4.1), the linear system arising from the discretization is known to be ill-conditioned, the condition number is essentially proportional to the reciprocal square of the mesh width [25]. Table 1 illustrates that attempting to solve the system in single precision fails, while double precision suffices to reduce the error according to finite element theory; and hence, to guarantee the result accuracy. In single precision, however, further refinement increases the error again: The additional refinement results in a solution that is objectively worse, although more unknowns are involved and hence more work is performed. Adding to the detriment, this behavior is difficult to notice without ground truth, because the solver still converges and reduces the residuals as expected.

2.4 Single vs. Double Precision Floating Point Performance

The relative performance improvement of single over double precision computations varies between architectures: For NVIDIA's GT200 GPU that we employ in our tests (cf. Section 4) the factor is eight, and five for AMD's R800 (Evergreen) chip. The first generation Cell processor [26] executed single precision floating point operations 14 times faster than double precision. Even on the SIMD units of conventional CPUs (SSE), single precision is twice as fast as double, the same factor has been announced for NVIDIA's new Fermi GPU [27]. In our applications, theoretical peak performance is not the relevant factor however, because we are mostly limited by the available bandwidth to off-chip memory (*memory wall problem*). The use of single precision consequently halves the bandwidth requirements of a given computation, and one can expect up to a twofold speedup.

The savings from using single precision are often even higher in practice, because the argument applies to all levels of storage: Single precision puts less pressure on the registers and twice the amount of data can be held in small, fast on-chip memories and caches, resulting in

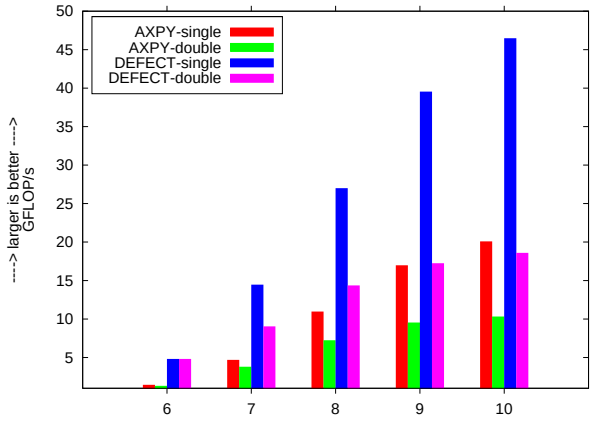

Fig. 2. Microbenchmarks in single and double precision for grid sizes $N = (2^L + 1)^2$ of varying level L.

a better data reuse and in particular in CUDA, a higher occupancy of multiprocessors.

We demonstrate this effect by two small microbenchmarks, executed on a GeForce GTX 280 GPU. The standard BLAS level 1 kernel `AXPY` ($\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$) is used to assess the savings for purely bandwidth-bound operations. In the `DEFECT` kernel ($\mathbf{d} = \mathbf{b} - A\mathbf{x}$) for a nine-banded matrix stemming from the underlying logical tensor product mesh, we use one CUDA thread per matrix row, and employ shared memory to manually cache portions of the coefficient vector $\mathbf{x}$ that are reused in the course of the computation, which additionally enables full coalescing and a minimal number of memory transactions. A second level of caching is provided by the on-chip texture cache, which is optimized for streaming locality. We refer to the references given in Section 1.1 for details on CUDA performance tuning strategies.

Figure 2 depicts the performance data we obtain for increasing level of refinement. The `AXPY` kernel is able to extract 85% and 87 % (120 GB/s and 123 GB/s) of the theoretical peak bandwidth for the largest input vectors in single and double precision respectively, and consequently, the GFLOP/s rate in double precision is half that of single precision as soon as the chip is entirely saturated. The carefully tuned implementation of the `DEFECT` kernel exploits data reuse via (manual) caching and results in excellent performance, reaching as much as 46.4 GFLOP/s in single precision and 18.5 GFLOP/s in double precision. So single precision is 2.5 times faster, more than the 2x factor we would expect from bandwidth considerations alone.

These numbers show the high potential of using single instead of double precision number formats. In Section 5 we show that these microbenchmark advantages indeed translate into a similar advantage for the complete multigrid solver. We achieve up to twice the performance while delivering results of equal accuracy (Table 5).

2.5 Mixed Precision Iterative Refinement

Mixed precision iterative refinement methods have been known for more than 100 years already. They gained rapid interest with the arrival of computer systems in the 1940s and 1950s. Wilkinson et al. [28], [29], [30] combined the approach with accumulated inner products as a mechanism to assess and increase the accuracy of computed results for linear system solvers, and provided a solid theoretical foundation of these methods. Shortly afterwards, Moler [31] extended the analysis from fixed point to floating point arithmetic.

The mixed precision solver for a linear system $A\mathbf{x} = \mathbf{b}$ basically comprises the following steps:

1) Compute $\mathbf{d} = \mathbf{b} - A\mathbf{x}$ in double precision.
2) Solve $A\mathbf{c} = \mathbf{d}$ approximately in single precision.
3) Update $\mathbf{x} = \mathbf{x} + \mathbf{c}$ in double precision.
4) Check for convergence and iterate.

For large sparse matrices step (2) uses an iterative solver. Therefore, we have two iterations: the *outer* iteration with the high precision updates and the *inner* iteration solving the low precision problem iteratively. Until the introduction of GPUs with native double precision support, the accurate solution of linear systems stemming from the discretization of PDE problems with finite elements was only possible with such a mixed precision scheme that executes step (2) on the GPU and the double precision outer loop on the CPU [9]: Native single precision yields inaccurate results (cf. Table 1), and as we have shown in the same paper, emulated arithmetic (double-single, storing low and high order portions of a quasi s46e8 number as an unevaluated sum of two single precision floating point values [32]) fails for the largest problem sizes. In this paper, we demonstrate that the mixed precision technique is superior on the GPU to executing entirely in double precision.

3 Multigrid Smoothers

After the discretization of the PDE we have a linear equation system $A_L\mathbf{x} = \mathbf{b}_L$ to solve where A_L is a sparse $N \times N$ matrix with nine bands, see Section 2.1. This is our problem on the finest level L. The multigrid cycle asks for the solution of $A_k\mathbf{x}_k = \mathbf{d}_k$ on each grid level, where A_k is the restriction of A_L to the k-th grid level and $\mathbf{d}_k$ the restriction of the defect from the previous level. All matrices A_k exhibit the nine band structure. So in the following discussion of the different options for smoothers for a fixed level k, we drop the index and simply seek to solve

$$A\mathbf{x} \quad = \quad \mathbf{d},$$

where A is a nine band matrix. Common to the following schemes is the iterative nature of the solution. Starting with some vector $\mathbf{x}^0$ we iterate

$$\mathbf{x}^{n+1} = F(\mathbf{x}^n)$$

for a fixed number of iterations (smoothing steps).

3.1 Jacobi

In case of the Jacobi smoother our iteration reads

$$\mathbf{x}^{n+1} = F_{\text{JAC}}(\mathbf{x}^n) \quad := \quad \mathbf{x}^n + \omega D^{-1}(\mathbf{d} - A\mathbf{x}^n),$$

where D is the diagonal of A and $\omega \in (0, 1]$ a damping factor. All components of $\mathbf{x}^{n+1}$ can be computed in parallel from $\mathbf{x}^n$. This simple smoother has been used by multigrid solvers on the GPU in the past, see the Introduction for references.

3.2 Gauß-Seidel

The Gauß-Seidel solver requires the update

$$\begin{aligned} \mathbf{x}^{n+1} = F_{\text{GS}}(\mathbf{x}^n) \quad &:= \quad (1-\omega)\mathbf{x}^n + \omega(L+D)^{-1}(\mathbf{d} - U\mathbf{x}^n), \\ A \quad &= \quad L + D + U, \end{aligned}$$

with the splitting of A into a strict lower triangular component L, the strict upper triangular matrix U and the diagonal D, and $\omega \in (0, 2)$. In case of $\omega > 1$ we speak of successive over-relaxation (SOR). The practical advantage of Gauß-Seidel over Jacobi is that typically, damping is not necessary. Here the update of $\mathbf{x}_\alpha^{n+1}$ depends on all $\mathbf{x}_\beta^{n+1}, \beta < \alpha$ for which $A_{\alpha\beta} = L_{\alpha\beta} \neq 0$, so they cannot be performed immediately in parallel.

In case of a 5-point stencil in 2D one typically uses a red-black Gauß-Seidel scheme to recover parallelism, because all odd-indexed unknowns can be updated independently of the even-indexed ones and vice versa. In our finite element setting we have a 9-point stencil and use a multi-coloring scheme with four colors $c_{00}, c_{01}, c_{10}, c_{11}$ to relax the dependencies. The coloring scheme assigns the color $c_{(x\%2, y\%2)}$ to the node with the index (x, y) in the 2D grid, so the nodes split into four disjoint index sets $C_{00}, C_{01}, C_{10}, C_{11}$ that correspond to the four colors: In other words, we alternate colors between lines in the mesh, and additionally alternate colors between adjacent degrees of freedom per line. The Gauß-Seidel update decomposes into four sequential in-place updates on these index sets, which are trivially parallel as described above (note that the operator F_{JAC} involves a defect calculation and thus incorporates previously computed contributions):

$$\begin{aligned} \mathbf{x}^{n+1} &= \mathbf{x}^n, \\ \forall \alpha \in C_{00} : \mathbf{x}_\alpha^{n+1} &= F_{\text{JAC}}(\mathbf{x}^{n+1})_\alpha, \\ \forall \alpha \in C_{01} : \mathbf{x}_\alpha^{n+1} &= F_{\text{JAC}}(\mathbf{x}^{n+1})_\alpha, \\ \forall \alpha \in C_{10} : \mathbf{x}_\alpha^{n+1} &= F_{\text{JAC}}(\mathbf{x}^{n+1})_\alpha, \\ \forall \alpha \in C_{11} : \mathbf{x}_\alpha^{n+1} &= F_{\text{JAC}}(\mathbf{x}^{n+1})_\alpha. \end{aligned}$$

3.3 Tridiagonal Solver

The disadvantage of the previous schemes is the weak coupling across the domain, i.e., it takes many iterations until the values near the left border of the domain influence the values near the right border, a characteristic of elliptic PDEs. To obtain a better coupling along one dimension we decouple the problem along the other dimension, i.e. we solve independently for all mesh lines l (these are not the matrix rows):

$$\forall \alpha_l \in I_l : (A\mathbf{x})_{\alpha_l} \quad = \quad \mathbf{d}_{\alpha_l},$$

where I_l is the set of indices belonging to the l-th line of nodes in the mesh. We can express the independent solves on the lines with the following matrix splitting,

$$\begin{aligned} &A = D + A^X + A^{\bar{X}}, \qquad (1) \\ &\forall \alpha_l \in I_l : ((D + A^X)\mathbf{x}^{n+1})_{\alpha_l} \;=\; (\mathbf{d} - A^{\bar{X}}\mathbf{x}^n)_{\alpha_l}, \\ &\mathbf{x}^{n+1} = F^X_{\text{TRIDI}}(\mathbf{x}^n) := \; (D + A^X)^{-1}(\mathbf{d} - A^{\bar{X}}\mathbf{x}^n), \end{aligned}$$

where D is the diagonal, A^X contains the coefficients that describe the interaction with the left and right nodes in the domain and $A^{\bar{X}}$ contains the remaining coefficients. We could apply an iterative scheme to solve these systems, but $(D+A^X)$ is a tridiagonal matrix which can be inverted efficiently by direct methods. On the CPU, the most commonly used scheme for this task is the Thomas algorithm, which is essentially Gaussian elimination for the case of a tridiagonal matrix [33].

Since the lines in the mesh are independent, we have enough parallel work to occupy all cores of a many-core device like the GPU. However, for an efficient inversion of each system we want to keep the required coefficients $(D + A^X)_{\alpha_l}$ and $(\mathbf{d} - A^{\bar{X}}\mathbf{x}^n)_{\alpha_l}$ in fast on-chip memory. For long mesh lines this means that only one line can be treated at the same time in each core. To use the parallelism in each core efficiently we thus require a parallel algorithm for solving Eq. 1, as the Thomas algorithm is inherently sequential. In the Introduction we have listed three such parallel algorithms. Cyclic reduction is the most common algorithm for this task, and it is work-efficient with a linear amount of arithmetic operations in the number of unknowns. The algorithm proceeds in two steps: a forward reduction and a backward substitution. Both steps can be performed in-place, which is beneficial in terms of the limited on-chip storage, but induces unfavorable memory access patterns that reduce the overall performance, see Figure 3. In contrast to previous GPU implementations of cyclic reduction we present a solution that is not completely in-place but performs much better because of more favorable memory access patterns, see Figure 4.

3.4 Cyclic Reduction on GPUs

In this section we describe our implementation of the cyclic reduction (CR) algorithm in detail. We have $M = \sqrt{N}$ lines in our mesh and in each line there are M equations e_α, $\alpha \in S^0 := \{0, \ldots, M-1\}$. As all mesh lines can be treated independently of each other, we describe the parallel solution within a single line. Let us define the three non-zero coefficients of the matrix and the right hand side corresponding to one mesh line as

$$\begin{aligned} (\mathbf{a}_\alpha^0, \mathbf{b}_\alpha^0, \mathbf{c}_\alpha^0) \quad &:= \quad (D + A^X)_\alpha, \qquad (2) \\ \mathbf{v}_\alpha^0 \quad &:= \quad (\mathbf{d} - A^{\bar{X}}\mathbf{x}^n)_\alpha \,. \end{aligned}$$

The CR algorithm reduces this $M \times M$ system to a $\lfloor M/2 \rfloor \times \lfloor M/2 \rfloor$ system with index set $S^1 := \{0, \ldots, \lfloor M/2 \rfloor - 1\}$ and recursively down to a 2×2 system. One thread per odd-indexed equation is used, and the reduction proceeds by computing for all odd $\alpha \in S^k$

$$\begin{aligned} f_l &:= \mathbf{a}^k_\alpha / \mathbf{b}^k_{\alpha-1}, \\ f_u &:= \mathbf{c}^k_\alpha / \mathbf{b}^k_{\alpha+1}, \\ \mathbf{a}^{k+1}_\alpha &:= -f_l \mathbf{a}^k_{\alpha-1}, \\ \mathbf{b}^{k+1}_\alpha &:= \mathbf{b}^k_\alpha - f_l \mathbf{c}^k_{\alpha-1} - f_u \mathbf{a}^k_{\alpha+1}, \\ \mathbf{c}^{k+1}_\alpha &:= -f_u \mathbf{c}^k_{\alpha+1}, \\ \mathbf{v}^{k+1}_\alpha &:= \mathbf{v}^k_\alpha - f_l \mathbf{v}^k_{\alpha-1} - f_u \mathbf{v}^k_{\alpha+1}. \end{aligned} \tag{3}$$

Each new equation is a linear combination of itself and its two even neighbors. After this update, we renumber all odd α from S^k consecutively and thus obtain the reduced index set S^{k+1}. This reduction continues until we reach a 2×2 system that can be solved directly, see Figure 3.

Then the backward substitution starts, which again proceeds recursively. Assume we reach the above situation and we have computed the solution to Eq. 1 in $\mathbf{v}^{k+1}_\alpha$ for all α in S^{k+1}. After renumbering, these indices correspond to the odd α in S^k. We obtain the solution for all even α from S^k with

$$\mathbf{v}^k_\alpha := (\mathbf{v}^{k+1}_\alpha - \mathbf{a}^{k+1}_\alpha \mathbf{v}^{k+1}_{\alpha-1} - \mathbf{c}^{k+1}_\alpha \mathbf{v}^{k+1}_{\alpha+1}) / \mathbf{b}^{k+1}_\alpha . \tag{4}$$

After this update we have the solution for both the odd and even α in the index set S^k, which means that we have the solution for all odd α in S^{k-1} and can continue with the backward substitution until in the last step we obtain the solution for the entire index set. Figure 3 visualizes this process.

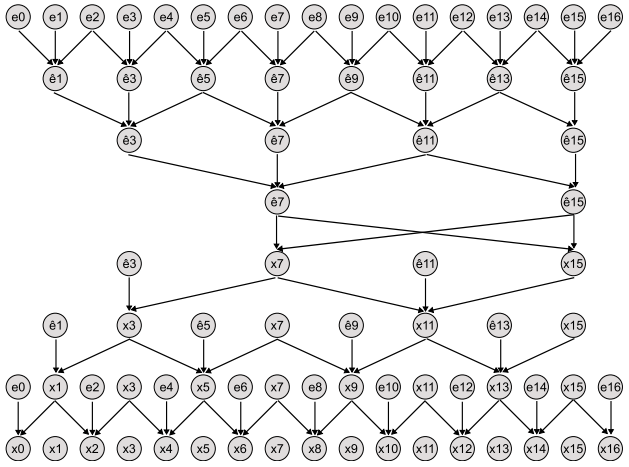

Fig. 3. Data flow and placement in shared memory in the standard cyclic reduction. Hats stand for updated equations and circles in the same column occupy the same memory address, i.e. denote in-place updates.

If all updates in the above scheme occur in-place, i.e. $(\mathbf{a}^k, \mathbf{b}^k, \mathbf{c}^k, \mathbf{v}^k)_\alpha$ all occupy the same storage for all k, then the reading and writing occurs with a 2^k stride in step k, see also Figure 3. This causes bank conflicts in the on-chip memory and greatly reduces the internal bandwidth: Shared memory in current GPUs is implemented with 16 memory banks, and if two concurrent threads address the same bank, accesses are serialized.

Our solution to this problem is to group the indices in each level of the reduction tree in two contiguous arrays of odd and even indices. When we load the initial data into shared memory (Eq. 2) we already separate the even and odd indices. With an appropriate padding between the arrays this is a bank conflict free read operation from global to shared memory. The output of each forward update step (Eq. 3) writes again into separate even and odd arrays. The even indices of level $k+1$ overwrite the location of the odd indices of level k in a contiguous fashion, whereas the odd indices of level $k+1$ are written into a new array, see Figure 4. With appropriate padding there are absolutely no bank conflicts in the involved read and write operations. As a tradeoff, we can remove the recursive padding and introduce 2-way conflicts for the writes only.

Unfortunately, any out-of-place CR scheme makes the backward substitution more difficult, because now the even-indexed $\mathbf{v}^k_\alpha$ cannot be updated in-place. Instead, they are updated as in Eq. 4 and written with a 2-stride into the the odd array on level k. The second 2-stride write into this array copies the already known odd-indexed $\mathbf{v}^k_\alpha$ values. Only on the last level 0 all even-indexed $\mathbf{v}^0_\alpha$ can be updated in place, before we write out the result back into global memory. Figure 4 depicts the data placement in this implementation. Because of the special treatment of the coarsest level the additional storage requirements are $M \cdot (1/4 + 1/8 + \ldots) = M/2$.

With 16 kB on-chip memory on current CUDA-capable GPUs we can thus solve a tridiagonal system with $M = 513$ (refinement level $L = 9$), as the storage requirement for all four vectors in single precision is $4 \cdot \frac{3}{2} \cdot 513 \cdot 4\,\mathrm{B} = 12,312\,\mathrm{B}$ plus few bytes of padding and local variables. We note that for the in-place algorithm $L = 9$ is also the maximum size, with shared memory requirements of 8,208 B. Accordingly, in double precision $M = 257$ ($L = 8$) is the current maximum. This underlines that the mixed precision approach discussed in this paper not only benefits faster transfers and computations, but very importantly enables fast on-chip solvers for larger problem sizes. Clearly, we could implement an algorithm for large M when all data does not fit into on-chip memory and we perform multiple transfers to global memory, but this variant would be significantly slower. Zhang et al. [18] report at least a 3x performance improvement when all updates are performed in shared memory instead of global memory. Our current implementation does not support larger problem sizes, as we focused on the improvements inside shared memory. However, the announced new Fermi GPU more than doubles the shared memory per multiprocessor, of which there will be fewer, however [27]. At most 48 kB of shared memory

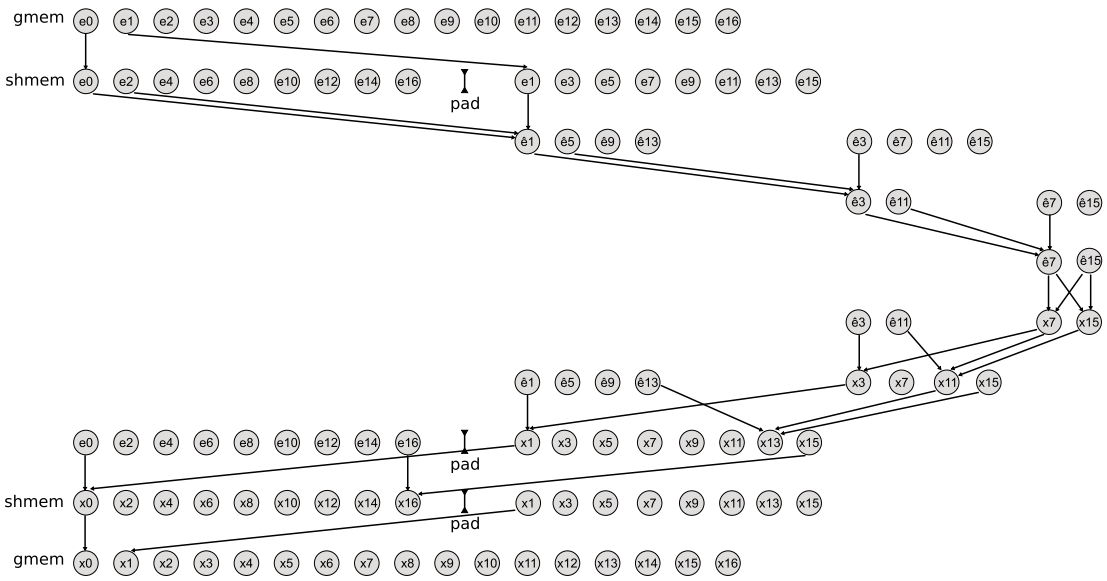

Fig. 4. Data placement in our implementation of cyclic reduction, hats stand for updated equations. Circles in the same column occupy the same memory address, i.e. in-place updates are performed. The data flow (arrow connections) is exactly the same as in Figure 3, but we have omitted most arrows to improve clarity.

is available, so we will be able to also solve the next problem size on level $L = 10$.

We compare the performance of our CR implementation with the best algorithm from the work by Zhang et al. [18] which clearly outperforms previous GPU implementations. Both Zhang et al. and we deal with the simultaneous solution of M tridiagonal systems with M unknowns each. In our case (due to the finite element discretization), M is always a power of two plus one, whereas Zhang et al. support powers of two, but the additional plus one does not make the code much more difficult. Zhang et al. report that the standard, in-place CR implementation on a GeForce GTX 280 GPU for 512 systems of 512 unknowns executes in 1.066 ms. Their fastest solver, a hybrid combination of CR and parallel cyclic reduction, which alleviates the disadvantages of the standard CR, needs less than half the time: 0.422 ms. In comparison our memory friendly CR implementation executes in 0.445 ms for 513 systems of 513-unknowns on the same GPU.

3.5 Alternating Direction Implicit Method

Section 3.3 described only solves along the lines of the mesh with the update operator F^X_{TRIDI} (Eq. 1). In this case the dependencies within each line are implicit but other dependencies are explicit. The same idea can be formulated for the columns of the domain obtaining the update operator F^Y_{TRIDI}, where the dependencies along the columns are implicit. If we alternate which of the directions is treated implicitely we obtain the alternating direction implicit (ADI, [34]) variant of the scheme. The complete ADI-TRIDI iteration then reads

$$\begin{aligned} \mathbf{x}^{n+1} &= F^X_{\text{TRIDI}}(\mathbf{x}^n), \\ \mathbf{x}^{n+2} &= F^Y_{\text{TRIDI}}(\mathbf{x}^{n+1}). \end{aligned}$$

In other words, the ADI-TRIDI smoother is able to capture anisotropies in both the horizontal and vertical direction.

Our implementation directly permutes the matrix bands after the assembly, and stores both row- and column oriented coefficients. Consequently, both sets of matrix bands are passed to the device. During the solution phase, the entries in the right hand side vector and the resulting solution vector need to be permuted from a row-based indexing to a column-based indexing and vice versa. This memory copy can be efficiently implemented analogously to a matrix transpose because of the underlying 2D mesh structure, and our implementation builds upon the corresponding example in the CUDA SDK (`transposeNew`).

4 Test Environment

All our tests are performed on a high-end workstation comprising an Intel Core2Duo E6750 CPU (Conroe, 4 096 kB level-2 cache, 2.66 GHz), and fast DDR2-800 memory in a dual-channel configuration (10.6 GB/s peak bandwidth). We execute our tests on only one core because a single subdomain is not parallelized across multiple cores in FEAST. For large problems FEAST schedules multiple subdomains as individual MPI jobs onto each cluster node until all cores are automatically occupied. We are convinced that a hybrid thread and MPI parallelization would not be of much benefit here, because on such coarse tasks the relative overhead of MPI processes is small and their intra-node communication already takes advantage of shared memory buffers in state-of-art MPI implementations, e.g. OpenMPI.

The GPU is an NVIDIA GeForce GTX 280 (GT200 chip), with 1 024 MB of GDDR3 memory connected via

a 512 bit bus for a theoretical peak memory bandwidth of 141.7 GB/s. The GPU features 30 multiprocessors, supports double precision, and is connected to the host system via PCIe x16 Gen2. The test system runs OpenSuSE 11.1 and the NVIDIA driver 190.18. All codes are compiled with the Intel C compiler (version 10.1), and we use the CUDA toolkit version 2.3.

We restrict ourselves to the 2D case because this is the configuration currently supported by FEAST. In general, the same techniques can be applied in the 3D case. The main difference for the 3D-ADI scheme is that now we alternate between three different 2D 'slice' variants, and each variant is again an alternation between the remaining two spatial directions.

4.1 Test Configurations

All experiments are performed for the same set of test cases as in our previous paper [9]. In addition to the dimensions of the domain, we also list the maximum aspect ratio on the finest level of refinement ($\mathrm{AR}_{\max}$) and the smallest mesh width $h_{\min}$. Furthermore, we compute the condition numbers of the system matrix for refinement levels $L \leq 6$ by brute force, using GNU Octave. This allows us to estimate the condition numbers for the larger problems, because they are proportional to the square of the mesh width. This estimate is not exact for the anisotropically refined test cases, but the computed values for smaller problem sizes give sufficient evidence that indeed $h_{\min}$ dominates the factor by which the condition number increases depending on the anisotropy factor ν.

- U1 No anisotropies: $\Omega = [0,1]^2$, uniform refinement, $\mathrm{AR}_{\max} = 1.0$, $h_{\min} =$ 9.77e-4 ($L = 10$). Condition numbers: 1.33e4 ($L = 8$), 5.31e4 ($L = 9$) and 2.12e5 ($L = 10$).
- U2 Weak uniform anisotropy: $\Omega = [0, 0.25] \times [0,1]$, uniform refinement, $\mathrm{AR}_{\max} = 4.0$, $h_{\min} =$ 2.44e-4 ($L = 10$). Condition numbers: 2.50e4 ($L = 8$), 1.00e5 ($L = 9$) and 4.00e5 ($L = 10$).
- U3 Medium uniform anisotropy: $\Omega = [0, 0.0625] \times [0,1]$, uniform refinement, $\mathrm{AR}_{\max} = 16.0$, $h_{\min} =$ 6.10e-5 ($L = 10$). Condition numbers: 2.70e4 ($L = 8$), 1.08e5 ($L = 9$) and 4.31e5 ($L = 10$).
- A1 Weak anisotropic refinement: $\Omega = [0,1]^2$, $\nu = 0.75$, $\mathrm{AR}_{\max} = 22.20$, $h_{\min} =$ 5.50e-5 ($L = 10$). Condition numbers: 7.20e4 ($L = 8$), 3.84e5 ($L = 9$) and 2.05e6 ($L = 10$).
- A2 Medium anisotropic refinement: $\Omega = [0,1]^2$, $\nu = 0.5$, $\mathrm{AR}_{\max} =$ 1.54e3, $h_{\min} =$ 9.54e-7 ($L = 10$). Condition numbers: 1.25e6 ($L = 8$), 1.00e7 ($L = 9$) and 8.00e7 ($L = 10$).
- A3 Hard anisotropic refinement: $\Omega = [0,1]^2$, $\nu = 0.25$, $\mathrm{AR}_{\max} =$ 1.84e6, $h_{\min} =$ 9.31e-10 ($L = 10$). Condition numbers: 2.11e8 ($L = 8$), 3.38e9 ($L = 9$) and 5.41e10 ($L = 10$). This test case yields a problem which cannot be assembled in single precision, because the Jacobi determinant of the mapping from the real, long and thin elements to the reference element evaluates to zero.
- A4 Hard anisotropic refinement: $\Omega = [0,1]^2$, $\nu = 0.0625$, $\mathrm{AR}_{\max} =$ 2.13e12, $h_{\min} =$ 8.88e-16 ($L = 10$). Condition numbers: 1.07e13 ($L = 8$), 6.83e14 ($L = 9$) and 4.37e16 ($L = 10$). This test case also yields a problem which cannot be assembled in single precision.
- A5 Extremely hard anisotropic refinement: $\Omega = [0,1]^2$, $\nu = 0.03125$, $\mathrm{AR}_{\max} =$ 2.08e15, $h_{\min} =$ 9.24e-19 ($L = 10$). Condition numbers: 2.57e15 ($L = 8$), 3.29e17 ($L = 9$) and 4.21e19 ($L = 10$). This test case can barely be assembled in double precision for refinement level 9, cannot be assembled for refinement level 10 in double precision, and is therefore extremely challenging for our mixed precision solver.

4.2 Solver Details

We use a multigrid solver for our tests, configured to perform four pre- and postsmoothing steps, and traversing the grid hierarchy in a V-cycle. We count one application of the ADI-TRIDI smoother as two elementary smoothing steps. All smoothers from Section 3 are used. A preconditioned conjugate gradient solver treats the coarse grid problems, its preconditioner is either Jacobi or ADI-TRIDI. The stopping criterion for the solver is set to reduce the initial residual by eight digits.

Two solvers execute entirely in double precision, either on the CPU or on the GPU. For the mixed precision solvers, the outer double precision correction loop now runs on the GPU, in contrast to our previous publication [9] when the CPU had to perform this task. The inner multigrid is configured to either perform one, two or three iterations, or to reduce the residual by one, two or three digits (in at most 32 cycles) before an outer update step is performed. Since the mixed precision scheme is beneficial on CPUs as well, we include it in the comparisons.

5 Results

5.1 Accuracy Studies

We perform all tests in this paragraph with the ADI-TRIDI smoother (preconditioner), as Jacobi- or Gauß-Seidel are not powerful enough for the challenging problems, or would require an impractically high number of smoothing steps (see also Section 5.4). We tabulate only levels 8 and 9 ($N = 257^2$ and $N = 513^2$ unknowns) to improve clarity of the presentation. As explained in Section 3.4, the double precision variant on the GPU requires too much shared memory to execute the larger $N = 513^2$ (L=9) problem size.

The l_2 errors (cf. Section 2.3) in Table 2 clearly show that neither the parallel solution on the GPU nor the application of a mixed precision scheme influence the accuracy of the final results. The final results of the six

different configurations of the mixed precision solver are also identical, so we only present one set of values. The same holds true for small differences between the mixed precision variant on the CPU, and the native double precision configuration on the GPU. The relative differences between the double and the mixed precision solvers are always below 1e-4, absolutely the solutions start to differ in the eighth decimal digit after the comma.

TABLE 2
l_2 errors for native and mixed precision solvers.

Test	Level	CPU-double	GPU-mixed
U1	8	1.7344895e-5	1.7344902e-5
U1	9	4.3362264e-6	4.3362292e-6
U2	8	1.6946217e-5	1.6946281e-5
U2	9	4.2365330e-6	4.2363011e-6
U3	8	1.6603963e-5	1.6603650e-5
U3	9	4.1508011e-6	4.1504573e-6
A1	8	2.2559231e-5	2.2559230e-5
A1	9	5.6398002e-6	5.6397099e-6
A2	8	3.3671244e-5	3.3671243e-5
A2	9	8.4177915e-6	8.4177905e-6
A3	8	4.9063089e-5	4.9063092e-5
A3	9	1.2265724e-5	1.2265727e-5
A4	8	6.3654794e-5	6.3654654e-5
A4	9	1.5913491e-5	1.5913506e-5
A5	8	6.6448219e-5	6.6447308e-5
A5	9	1.6612151e-5	1.6612856e-5

5.2 Performance of Mixed Precision Variants

Next, we examine the performance of the six mixed precision variants, both in terms of numerical efficiency, i.e., the relation between the number of inner and outer iterations depending on the stopping criterion of the inner solver, and in terms of absolute runtime. Timings are given in seconds, and we restrict ourselves again to the ADI-TRIDI smoother. The systems are assembled in double precision on the CPU, and are then solved either on the CPU or on the GPU. All timings include the initial transfer of the right hand side to the device, and the readback of the solution vector to the CPU, but do not include the transfer of the matrix, as this is the relevant situation for our CSM and CFD applications (cf. Section 2.2): The cost of the matrix transfer is typically amortized in practice because the systems are solved for many right hand sides.

A number of important conclusions can be drawn from the results in Tables 3 and 4. Overall, the timings are consistent: The more outer loop iterations / update steps are performed, the less do the total iteration numbers deviate from the reference values of the CPU double precision solver. On the other hand, many outer loop iterations tend to increase the fraction of the slower double precision computations, slightly deteriorating runtime performance. We observe a general trend in the results from Table 3 and Table 4: The two configurations that perform exactly one multigrid cycle or gain one digit in each outer loop iteration are slower than the variants that perform two steps or gain two digits. Finally, performing too much work in the inner solver is not always beneficial. The fixed iteration variant does not suffer, because the same total number of inner iterations are spread over less global updates, resulting in slightly faster performance. Gaining three digits by the inner solver however leads to a significant increase in the iteration numbers, in particular for the less ill-conditioned test cases: The inner solver occasionally stalls, the concrete numbers are a consequence of setting the maximum admissible number of inner multigrid cycles per solver call to 32. While the number of update steps goes down as expected, the additional inner work outweighs this advantage.

TABLE 3
Performance of mixed precision variants on the GPU: Fixed number of iterations (Notation 'a:b': 'a' equals the number of outer iterations, 'b' equals the total sum of inner iterations).

		double	1 iter		2 iter		3 iter	
ID	L	#it	#it	time	#it	time	#it	time
U1	8	5	5:5	0.0345	3:6	0.0400	3:9	0.0586
U1	9	5	5:5	0.0659	3:6	0.0765	3:9	0.1131
U2	8	5	6:6	0.0434	3:6	0.0379	3:9	0.0577
U2	9	6	6:6	0.0809	3:6	0.0763	3:9	0.1124
U3	8	5	5:5	0.0038	3:6	0.0390	3:9	0.0577
U3	9	4	5:5	0.0658	3:6	0.0760	3:9	0.1147
A1	8	6	5:5	0.0354	3:6	0.0458	3:9	0.0674
A1	9	6	6:6	0.0790	3:6	0.0778	3:9	0.1136
A2	8	6	6:6	0.0434	3:6	0.0401	2:6	0.0396
A2	9	6	6:6	0.0790	3:6	0.0766	3:9	0.1189
A3	8	6	6:6	0.0411	3:6	0.0403	3:9	0.0587
A3	9	6	6:6	0.0787	3:6	0.0765	3:9	0.1129
A4	8	8	8:8	0.0554	4:8	0.0527	3:9	0.0586
A4	9	9	8:8	0.1055	5:10	0.1273	3:9	0.1128
A5	8	10	11:11	0.0754	6:12	0.0792	4:12	0.0782
A5	9	11	11:11	0.1449	6:12	0.1524	4:12	0.1505

TABLE 4
Performance of mixed precision variants on the GPU: Gaining digits (Notation 'a:b': 'a' equals the number of outer iterations, 'b' equals the total sum of inner iterations).

		double	1 digit		2 digits		3 digits	
ID	L	#it	#it	time	#it	time	#it	time
U1	8	5	5:5	0.0363	3:6	0.0406	3:6	0.0513
U1	9	5	5:5	0.0665	4:7	0.0901	3:36	0.4500
U2	8	5	6:6	0.0433	4:7	0.0472	3:7	0.0461
U2	9	6	6:6	0.0816	4:7	0.0906	3:37	0.4551
U3	8	5	5:5	0.0344	3:5	0.0333	3:7	0.0458
U3	9	4	5:5	0.0664	3:5	0.0649	3:37	0.4657
A1	8	6	5:5	0.0352	3:6	0.0418	3:7	0.0407
A1	9	6	6:6	0.0802	3:6	0.0792	3:36	0.4509
A2	8	6	6:6	0.0418	3:6	0.0408	2:7	0.0466
A2	9	6	6:6	0.0796	3:6	0.0772	3:37	0.4614
A3	8	6	6:6	0.0416	3:6	0.0412	3:7	0.0469
A3	9	6	6:6	0.0799	3:6	0.0768	3:9	0.1142
A4	8	8	5:8	0.0543	3:8	0.0528	3:11	0.0717
A4	9	9	5:8	0.1034	4:11	0.1392	3:11	0.1378
A5	8	10	6:11	0.0737	4:13	0.0986	3:15	0.0972
A5	9	11	6:11	0.1420	4:13	0.1636	3:14	0.1743

This is not a big surprise, because single precision is insufficient to *solve* these problems, and reducing the residual by three digits constitutes almost 40% of the solution process that requires eight digits total accuracy.

We note that switching from refinement level 8 to 9, i.e. quadrupling the problem size, increases runtime to solution much more favorably, see Section 2.4 for associated microbenchmarks highlighting the saturation of the device, and also Section 5.4 for computations on level 10.

5.3 Speedup Measurements

On the CPU, tridiagonal solves are implemented using the Thomas algorithm, which is essentially Gaussian elimination and requires $6M$ operations for a system with M unknowns. Cyclic reduction is also linear in the number of unknowns, but more expensive and performs approximately $23M$ arithmetic operations. Our speedup comparisons are fair in the sense that we do not compare against an algorithmically suboptimal CPU implementation but rather against a carefully tuned more efficient variant, and in particular because we continue to include the bus transfers of the right hand side and the solution into our measurements.

TABLE 5
Speedup of mixed precision variants over double precision solvers on the CPU and the GPU. Abbreviations: 'C(d)' denotes double precision on the CPU, 'G(m)' mixed precision on the GPU, etc.

	L	C(d)/G(d)	C(m)/G(m)	C(d)/G(m)	G(d)/G(m)
U1	8	3.25	3.89	5.41	1.66
U1	9		9.69	12.21	
U2	8	3.10	3.77	5.49	1.77
U2	9		9.66	14.72	
U3	8	3.15	3.78	5.28	1.67
U3	9		9.64	17.04	
A1	8	3.26	3.75	5.27	1.62
A1	9		9.48	16.66	
A2	8	3.26	3.64	6.49	1.99
A2	9		9.60	17.21	
A3	8	3.25	3.76	6.38	1.96
A3	9		9.82	17.58	
A4	8	3.18	3.67	6.56	2.06
A4	9		8.76	12.62	
A5	8	2.91	3.17	4.39	1.51
A5	9		9.58	14.77	

Table 5 exemplarily presents four different speedup factors, and we restrict the discussion to the mixed precision variant that gains two digits in the inner loop, because this configuration performs on average best for the entire set of test problems. We first compare executing entirely in double precision on the CPU and the GPU. Then, we make the same comparison for the respective mixed precision solvers. As the typical situation in many legacy CPU codes is to use double precision exclusively, we also compute the speedup of the mixed precision GPU solver over this configuration. Finally, we quantify the advantage of the mixed precision scheme on the GPU alone. To avoid granularity effects (one additional iteration reduces the residuals by more than the prescribed eight digits), we normalize all timings to the time per unknown to gain exactly one digit, $T/(N \cdot \#\text{iters} \cdot \rho)$, ρ denotes the convergence rate of the solver. Table 6 lists the speedup factors computed from these normalized timings. Such normalization is always beneficial, because now the timings are independent of the problem size and the problem at hand.

TABLE 6
Speedup of mixed precision variants over double precision solvers on the CPU and the GPU. The data is the same as in Table 5, but the speedups are calculated from normalized time per unknown for gaining one digit.

	L	C'(d)/G'(d)	C'(m)/G'(m)	C'(d)/G'(m)	G'(d)/G'(m)
U1	8	3.25	3.92	6.38	1.96
U1	9		9.71	16.43	
U2	8	3.10	3.77	6.26	2.02
U2	9		9.72	16.94	
U3	8	3.13	3.72	5.59	1.78
U3	9		9.36	15.09	
A1	8	3.26	3.76	6.17	1.89
A1	9		9.36	15.94	
A2	8	3.26	3.64	6.45	1.98
A2	9		9.62	16.95	
A3	8	3.24	3.74	6.25	1.93
A3	9		9.79	16.78	
A4	8	3.17	3.60	6.44	2.03
A4	9		8.93	15.22	
A5	8	3.03	3.11	5.16	1.70
A5	9		9.56	16.97	

In double precision, the speedups are consistently larger than a factor of three, on the GPU, double precision computations are limited to refinement level $L = 8$, where the device is not entirely saturated. For this problem size, only one thread block is concurrently active per multiprocessor due to the limited size of the shared memory, and thus, the ability to hide stalls due to off-chip memory accesses by switching to other warps is not fully exploited. In fact, the larger problem size can currently only be treated efficiently with a mixed precision approach. When comparing the conventional double precision CPU implementation with the mixed precision GPU solver, we observe speedups of more than a factor of 15 for the largest admissible problem size. The CPU becomes less efficient when moving from refinement level $L = 8$ to refinement level $L = 9$ as more data has to be moved in and out of the limited level-2 cache more frequently. On the GPU, it is the other way round, which is underlined by the significantly better speedups on the larger problem sizes. With the new Fermi chip, we can fit an even larger cyclic reduction into shared memory, and expect even better speedups on this GPU, see Section 3.4. The speedup factors are smaller in the fairer comparison of the two mixed precision solvers, but still reach almost an order of magnitude, which clearly highlights the advantages of the GPU. Finally,

at least a 70% speedup is achieved on the device alone by the mixed precision scheme, often reaching a factor of two.

5.4 Smoother Comparison

In this experiment, we want to demonstrate that different problems require different smoothers for optimal performance. We evaluate the performance of all smoothers presented in Section 3, first for the purely isotropic U1 test case, and then for the anisotropic A2 one. We employ the mixed precision solver that gains two digits in the inner loop, and execute it on the CPU and on the GPU. All measurements are normalized to time per unknown per digit, so smaller values are better.

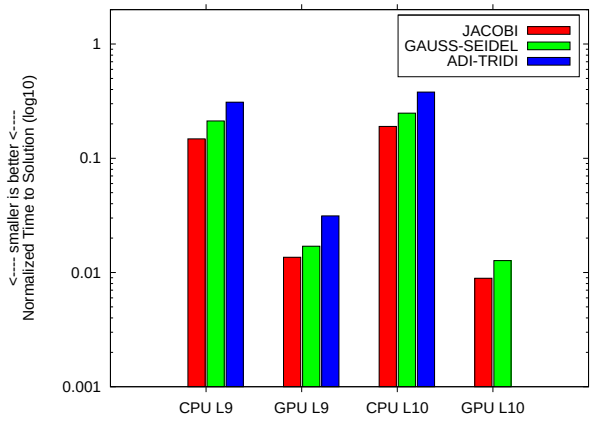

Fig. 5. Comparison of multigrid performance with different smoothers, U1 configuration, logarithmic scale on the y-axis.

Figure 5 depicts the results for the purely isotropic test domain (U1). The simple Jacobi smoother (damped with $\omega = 0.7$) is always the most efficient one. For the two refinement levels, the GPU solver is 10.9 and 21.2 times faster than the CPU variant, respectively. It is important to note that in this metric, which is independent of the problem size, the importance of device saturation can be seen clearly, the GPU is faster on the larger problem size than on the smaller one. In absolute numbers, the GPU solver executes in 0.1 seconds for $L = 10$. The Gauß-Seidel smoother yields slightly larger timings, but the speedups are still 12.5 and 19.5. The ADI-TRIDI smoother is the least efficient one, and takes approximately 2.3 times longer for this simple test problem.

For the harder, anisotropic test problem however (see Figure 6), the ADI-TRIDI smoother is obligatory, and leads to the shortest time to solution. The Jacobi smoother (strong damping of $\omega = 0.6$ is required to ensure convergence) is not applicable for all practical purposes, and takes more than 14 times longer to solve the problem than multigrid equipped with the stronger smoother. Gauß-Seidel performs slightly better than Jacobi, but remains clearly inferior to ADI-TRIDI.

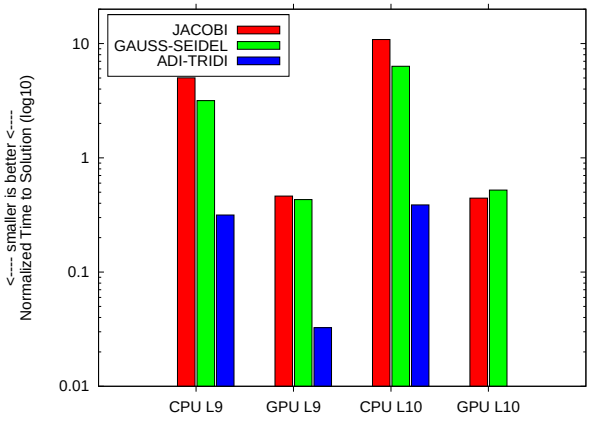

Fig. 6. Comparison of multigrid performance with different smoothers, A2 configuration, logarithmic scale on the y-axis.

6 Conclusions

Our goal has been to design an efficient and robust GPU-multigrid solver that can solve large, sparse linear equation systems stemming from PDE discretizations on highly anisotropic grids. This is a crucial component in our framework for large scale simulations on GPU clusters, but also highly relevant for other standalone applications. For this purpose we have developed a high performance cyclic reduction implementation that exhibits much better memory access patterns than the standard implementation. Accordingly, it performs significantly better, on a par with the best hybrid tridiagonal GPU solvers published so far. We use this solver as a smoother in a multigrid cycle, and as a preconditioner in a Krylov subspace scheme, and are thus able to solve linear equation systems on grids with high anisotropies. A comparison of multigrid performance with different smoothers shows that our multi-color Gauß-Seidel scheme is a faster choice for almost-isotropic grids, but only the new ADI-TRIDI powered multigrid can deal with the more difficult and realistic test cases. Finally, we obtain further acceleration of 70% and more for the entire solution process by running our previously developed mixed-precision iterative refinement now completely on the GPU. In comparison to the CPU the speedups are close to an order of magnitude, and consistenty above 15 when comparing against double precision on the CPU alone.

Acknowledgments

We would like to thank Stefan Turek, Hilmar Wobker, Sven H.M. Buijssen, Dirk Ribbrock, Markus Geveler and the FEAST team at TU Dortmund for support. John D. Owens, Yao Zhang and Jonathan Cohen provided invaluable insight into the odds and ends of tridiagonal solvers on GPUs, and the discussions of their work compared to ours were very fruitful. Thanks to NVIDIA

for generous hardware donations, and to Mark Harris for useful comments during performance tuning. This work has been partly supported by the DFG in projects TU102/22-1, 22-2; and the BMBF ('HPC Software für skalierbare Parallelrechner'), in the SKALB project (01IH08003D / SKALB).

REFERENCES

[1] NVIDIA Corporation, "NVIDIA CUDA programming guide version 2.3," Jul. 2009, http://www.nvidia.com/cuda.
[2] J. D. Owens, M. Houston, D. P. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[3] M. Garland, S. L. Grand, J. Nickolls, J. A. Anderson, J. Hardwick, S. Morton, E. H. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008.
[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar./Apr. 2008.
[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, Mar./Apr. 2008.
[6] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 917–924, Jul. 2003.
[7] N. Goodnight, C. Woolley, G. Lewin, D. P. Luebke, and G. Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Graphics Hardware 2003*, M. Doggett, W. Heidrich, W. R. Mark, and A. Schilling, Eds., Jul. 2003, pp. 102–111.
[8] R. Strzodka, M. Droske, and M. Rumpf, "Image registration by a regularized gradient flow – A streaming implementation in DX9 graphics hardware," *Computing*, vol. 73, no. 4, pp. 373–389, Nov. 2004.
[9] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, Jan. 2007.
[10] M. Kazhdan and H. Hoppe, "Streaming multigrid for gradient-domain operations on large images," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1–10, Aug. 2008.
[11] Z. Feng and P. Li, "Multigrid on GPU: Tackling power grid analysis on parallel SIMT platforms," in *ICCAD 2008: IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2008, pp. 647–654.
[12] E. Elsen, P. LeGresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, vol. 227, no. 24, pp. 10 148–10 161, Dec. 2008.
[13] M. Kass, A. E. Lefohn, and J. D. Owens, "Interactive depth of field using simulated diffusion," Pixar Animation Studios, Tech. Rep. 06-01, Jan. 2006.
[14] S. Sengupta, M. J. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Graphics Hardware 2007*, T. Aila and M. Segal, Eds., Aug. 2007, pp. 97–106.
[15] R. W. Hockney, "A fast direct solution of Poisson's equation using Fourier analysis," *Journal of the ACM*, vol. 12, no. 1, pp. 95–113, Jan. 1965.
[16] R. W. Hockney and C. R. Jesshope, *Parallel Computers*. Adam Hilger, Nov. 1981.
[17] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," *Journal of the ACM*, vol. 20, no. 1, pp. 27–38, Jan. 1973.
[18] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, Jan. 2010, pp. 127–136.
[19] M. Grajewski, M. Köster, and S. Turek, "Mathematical and numerical analysis of a robust and efficient grid deformation method in the finite element context," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 1539–1557, Jan. 2009.
[20] S. Turek, C. Becker, and S. Kilian, "Hardware–oriented numerics and concepts for PDE software," *Future Generation Computer Systems*, vol. 22, no. 1-2, pp. 217–238, Feb. 2004.
[21] S. Turek, D. Göddeke, C. Becker, S. H. Buijssen, and H. Wobker, "FEAST – Realisation of hardware-oriented numerics for HPC simulations with finite elements," *Concurrency and Computation: Practice and Experience*, Feb. 2010, special Issue Proceedings of ISC 2008.
[22] S. Turek, C. Becker, S. Kilian, S. H. M. Buijssen, D. Göddeke, and H. Wobker, "FEAST – finite element analysis and solution tools," 2008, http://www.feast.tu-dortmund.de.
[23] D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. S. McCormick, and S. Turek, "Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU," *International Journal of Computational Science and Engineering*, vol. 4, no. 4, pp. 254–269, Oct. 2009.
[24] D. Göddeke, S. H. Buijssen, H. Wobker, and S. Turek, "GPU acceleration of an unmodified parallel finite element Navier-Stokes solver," in *High Performance Computing & Simulation 2009*, W. W. Smari and J. P. McIntire, Eds., Jun. 2009, pp. 12–21.
[25] O. Axelsson and V. A. Barker, *Finite Element Solution of Boundary Value Problems*, ser. Classics in Applied Mathematics. SIAM, Jul. 2001, vol. 35.
[26] D. C. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. R. Johns, J. A. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. L. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *Solid-State Circuits Conference, ISSCC 2005, Digest of Technical Papers*, Feb. 2005, pp. 184–592 Vol. 1.
[27] NVIDIA Corporation, "Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi," Sep. 2009, http://www.nvidia.com/object/fermi_architecture.html.
[28] J. H. Wilkinson, *Rounding errors in algebraic processes*. Prentice-Hall, Jan. 1963.
[29] R. S. Martin, G. Peters, and J. H. Wilkinson, "Iterative refinement of the solution of a positive definite system of equations," *Numerische Mathematik*, vol. 8, no. 3, pp. 203–216, May 1966.
[30] H. J. Bowdler, R. S. Martin, G. Peters, and J. H. Wilkinson, "Solution of real and complex systems of linear equations," *Numerische Mathematik*, vol. 8, no. 3, pp. 217–234, May 1966.
[31] C. B. Moler, "Iterative refinement in floating point," *Journal of the ACM*, vol. 14, no. 2, pp. 316–321, Apr. 1967.
[32] D. E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, Oct. 1997.
[33] L. H. Thomas, "Elliptic problems in linear difference equations over a network," 1949, Watson Scientific Computing Laboratory Report, Columbia University, New York.
[34] D. W. Peaceman and H. H. Rachford Jr., "The numerical solution of parabolic and elliptic differential equations," *Journal of the Society for Industrial and Applied Mathematics*, vol. 3, no. 1, pp. 28–41, Mar. 1955.

Dominik Göddeke recently received his PhD degree in Applied Mathematics at TU Dortmund, Germany, under the supervision of Stefan Turek. His research interests cover hardware-oriented numerics in computational science and engineering, including parallelization, large-scale computations, high-performance computing, multigrid methods, finite element techniques and domain decomposition. His main focus is on using GPUs for these tasks, and he has published a range of papers about GPU Computing since 2005.

Robert Strzodka is the head of the research group Integrative Scientific Computing at the Max Planck Institute for Computer Science in Saarbrücken, Germany since 2007. His research focuses on efficient interactions of mathematic, algorithmic and architectural aspects in heterogeneous high performance computing. Previously, Robert was a visiting assistant professor in computer science at the Stanford University and until 2005 a postdoc at the Center of Advanced European Studies and Research in Bonn. He received his doctorate in numerical mathematics from the University of Duisburg-Essen in 2004.