Increasing Arithmetic Intensity in Multigrid Methods on GPUs Using Block Smoothers

Matthias Bolten[1(✉)] and Oliver Letterer[2]

[1] Institut für Mathematik, Universität Kassel, Heinrich-Plett-Straße 40, 34132 Kassel, Germany
bolten@mathematik.uni-kassel.de

[2] Fakultät für Mathematik und Naturwissenschaften, Bergische Universität Wuppertal, 42097 Wuppertal, Germany

Abstract. Block smoothers that relax small blocks of unknowns provide a way to increase the amount of arithmetic operations needed per smoothing iteration. If the block sizes are small, the variables associated to these blocks fit in fast local memories, thus allowing for a better exploitation of modern computer architectures. At the same time block smoothers are efficient smoothers that allow for more aggressive coarsening resulting in less coarse grids.

We implemented block smoothers in combination with aggressive coarsening in OpenCL, targeting GPUs. Two different data layouts were compared for the smoother. The results show that while the more advanced data layout does not yield a better performance, the introduction of block smoothers in multigrid methods can indeed reduce the time to solution on a GPU.

Keywords: Multigrid methods · High arithmetic intensity · Block smoothers · OpenCL · GPU

1 Introduction

In many simulations in computational science and engineering linear systems

$$Ax = b, A \in \mathbb{R}^{n \times n},\ x, b \in \mathbb{R}^n,$$

have to be solved, often these linear systems arise from the discretization of partial differential equations (PDEs). Frequently the overall run time of the simulation is governed by the time spent in the solver, therefore there is a huge demand for optimal scalable solvers. If the number of unknowns is large the direct solution of the system is usually not feasible, so iterative methods are used. In many cases, especially when the underlying PDE is elliptic, multigrid methods are known to be optimal iterative solvers, i.e., the convergence rate does not depend on the system dimension. Further, multigrid methods can be parallelized efficiently and scale to huge numbers of processors. The computational complexity that determines the cost of one multigrid cycle is governed by the number of unknowns per row in the system matrix. In the case of the usually used low-order discretization of a PDE independently of wether a finite difference, finite volume or finite element scheme is used, this number is usually small. As a consequence the number of arithmetic operations that has to be carried out per unknown and thus per memory transfer is low. This limits the achievable performance of multigrid methods in terms of FLOPS compared to the theoretically achievable performance on modern computer architectures. While this limit exists for a long time, in recent years it has become more prominent as the performance of processors increases much faster than the performance of memory does.

On parallel computers the scalability of multigrid methods is limited only through the necessary global information exchange that is inherently necessary to solve the problems at hand due to their global nature. This requirement introduces a logarithmic dependency of the runtime on the number of processors, the degrade in scalability mostly is due to the relatively low amount of work that has to be conducted on the lower levels compared to the communication. One way to hide this effect for larger processor numbers is the use of aggressive coarsening that results in a lower number of levels in the multigrid hierarchy. For multigrid methods employing aggressive coarsening to be as effective as standard multigrid methods more powerful smoothers are needed. This can either be accomplished by carrying out more smoothing steps or by using completely different smoothers, e.g., polynomial smoothers.

In order to overcome both limitations we propose to use block smoothers that result in a higher arithmetic cost than usually employed point smoothers but at the same time reduce high frequency components of the error more effectively. In contrast to line and plane smoothers that are often used in multigrid methods, e.g., when anisotropies are present in the underlying PDE, by block smoothers we generally entitle methods that do not relax the residual at one point at a time but at a set of points concurrently. In the cases considered here these sets will be local subdomains. If the variables are stored in memory appropriately this results in a high locality of the data that is used during a relaxation and thus less memory transfers are needed. Even if the arithmetic cost per unknown is higher than in point smoothers, the combination of a better smoothing factor together with the better use of modern architectures results in an overall reduced time to solution.

In this paper we present an implementation of block smoothers on GPUs using OpenCL to showcase that our proposal is feasible. In fact combining aggressive coarsening with block smoothers on a GPU results in a multigrid method that has a lower convergence rate but a reduced time to solution although more iterations of the multigrid method are necessary.

The rest of the paper is structured as follows: In the next section we will provide a brief introduction into multigrid methods. In Sect. 3 the block smoothers used here are presented and afterwards in the following section the implementation on the GPU is described. Numerical results are shown in Sect. 5 and the paper closes with a conclusion and outlook.

Algorithm 1. Multigrid cycle $x_{n_i} = \mathcal{MG}_i(x_{n_i}, b_{n_i})$

```
$x_{n_i} \leftarrow \mathcal{S}_i^{\nu_1}(x_{n_i}, b_{n_i})$
$r_{n_i} \leftarrow b_{n_i} - A_i x_{n_i}$
$r_{n_{i+1}} \leftarrow R_i r_{n_i}$
$e_{n_{i+1}} \leftarrow 0$
if $i + 1 = l_{\max}$ then
  $e_{n_{l_{\max}}} \leftarrow A_{l_{\max}}^{-1} r_{n_{l_{\max}}}$
else
  for $j = 1, \dots, \gamma$ do
    $e_{n_{i+1}} \leftarrow \mathcal{MG}_{i+1}(e_{n_{i+1}}, r_{n_{i+1}})$
  end for
end if
$e_{n_i} \leftarrow P_i e_{n_{i+1}}$
$x_{n_i} \leftarrow x_{n_i} + e_{n_i}$
$x_{n_i} \leftarrow \tilde{\mathcal{S}}_i^{\nu_2}(x_{n_i}, b_{n_i})$
```

2 Multigrid Methods

Multigrid methods go back to [1,6,7], their use in applications has been promoted in [3,11,12]. In the following we describe geometric multigrid methods that are based on the following observation: When an iterative method like Gauß-Seidel or damped Jacobi is applied to a linear system that arises from the discretization of a simple elliptic PDE like the Poisson equation and when plotting the error on the discretization grid before and after a few steps of the iterative method it is much smoother after the application. As a consequence it will be well-represented on a coarser grid where the problem is less expensive to solve. This idea is applied recursively resulting in the so–called V-cycle, if multigrid is called multiple times recursively, the result are other cycling schemes. In Algorithm 1 the basic multigrid algorithm is provided. In addition to the different grids that are needed to represent the approximation to the solution on the various levels, smoothers and grid transfer operators have to be defined. As smoothers usually point smoothers like the aforementioned Gauß-Seidel method or damped Jacobi are used. Other options include polynomial smoothers, incomplete factorizations or block smoothers like the ones defined in Sect. 3. Based on the observation of the behavior of the error as grid transfer operators methods like linear or higher order interpolation are used to transfer the error from the coarse to the fine level, in the opposite direction the simplest option is injection of the fine level solution, i.e., the current value at a grid point is just copied to the coarse level. Another option that is often used is full-weighting that is the transpose of the linear interpolation, possibly multiplied with a scalar factor. Extensions to multigrid methods that do not require an a priori defined grid hierarchy are known as algebraic multigrid (AMG). An introduction to geometric multigrid methods can be found in [4], more details and an introduction to AMG are found in [16].

The efficient implementation of parallel multigrid has been discussed in different papers, an overview over parallelization of multigrid is provided in [5], here also the aggressive coarsening that is used here is presented. Multigrid methods for GPUs have been presented before, e.g., in [8–10].

A geometric multigrid method starts from a given partial differential equation

$$\begin{aligned} \mathcal{L}u &= f, \qquad \text{in } \Omega \\ u &= g, \qquad \text{in } \partial\Omega. \end{aligned}$$

Other boundary conditions than Dirichlet boundary conditions are possible. The equation is discretized using different discretization parameters h resulting in linear systems of the form

$$L_h u_h = f_h,$$

where the boundary conditions are eliminated to be contained in L_h or they are handled explicitly. Grid transfer operators are defined to transfer quantities between the different levels of discretization in a geometrically motivated manner and simple iterative schemes like Gauß-Seidel are added as described before in Algorithm 1 to obtain a multigrid method.

Here, we limit ourselves to cuboidal domains discretized using regular grids. In the simplest case of the unit cube discretized with n grid points in each direction we end up with a linear system with n^3 unkowns. The coarser levels are obtained by subsequently taking every gth grid point, only. Usually, a coarsening ratio of $g = 2$ is chosen and the grid sizes are chosen such that we end up with 1 unkown, only. In the case of Dirichlet boundary conditions this results in $n = 2^k - 1$ grid points while for periodic boundary conditions we obtain $n = 2^k$ grid points. As interpolation linear interpolation is used, i.e., the value at a fine grid point is taken as the weighted average between neighboring points. The interpolation is taken as full-weighting with the same weights, i.e., the transpose of the restriction operator.

3 Block Smoothers

Point smoothers have a relatively small number of arithmetic operations per memory transfer, resulting in a poor use of modern processor architectures. At the same time, when aggressive coarsening is employed the smoothing factor drops substantially.

In [2] the usage of block smoothers in multigrid methods has been proposed and preliminary results of analyzing block smoothers using local Fourier analysis were given. These block smoothers are using a domain decomposition approach, i.e., the unknowns are partitioned into smaller sets forming a connected subdomain Ω_i of the whole domain under consideration. An introduction to domain decomposition can be found in [14]. The union of the subdomains is the whole domain, i.e.,

$$\bigcup_i \Omega_i = \Omega,$$

the subdomains do not have to be disjoint. One step of the block smoother consists of a loop over the subdomains. Within the loop the residual is calculated, the linear system is being restricted to the variables corresponding to the current subdomain, the restricted system is solved for the restricted residual as right hand side and finally the current guess is updated by adding the result of this small system. This results in a relaxation of a whole subdomain instead of an individual variable. This method is known as block Gauß-Seidel or multiplicative Schwarz. If the residual is not updated within the loop over the domains but rather just once before, the method is a block Jacobi-method or additive Schwarz method. It is known from the underlying theory that multiplicative methods work better than additive, just like in the scalar case [14]. The subdomains can be chosen on each level individually. As this is used as a smoothing procedure it is further not necessary to solve each of the restricted systems exactly, but rather using an iterative method. When small block sizes are chosen even a plain Gauß-Seidel method is well-suited for this task, as its convergence factor depends on the ratio of the grid spacing and the domain size that will be quite large in this case. As in the point relaxation case a lexicographic ordering of the blocks results in a method that is inherently sequential. This is not the case for block Jacobi-type methods, but the smoothing factor of block Jacobi is worse than that of block Gauß–Seidel. By using a multicoloring of the blocks also the Gauß–Seidel variant of the smoother is parallelizable.

As the solution of the restricted linear systems is much more expensive than the relaxation of an individual variable, the overall method is more expensive. On the other hand, the resulting methods are much more efficient as smoothers than point-relaxation methods. If the subdomains are relatively small, the overhead introduced by the method is relatively small, as well. Further, if a memory layout is chosen that keeps all needed variables in the cache, the solution of the linear system and thus the relaxation of one block can be calculated very fast as modern processors can be used more efficiently. This is true for direct solvers that are used to solve the restricted linear system as well as for iterative solvers, as both will benefit from the advantageous memory layout.

As we are dealing with cuboidal domains, we consider cuboidal subdomains, as well. To allow for parallel processing in the smoother a multi-coloring scheme of the blocks is employed.

4 Implementation

A multigrid method for cuboidal domains with equispaced regular grids was implemented in OpenCL to measure the performance gain of the proposed method on GPUs. Parameters like the work-group size were set to be chosen automatically by OpenCL. The multigrid method uses aggressive coarsening to reduce the number of levels that is present, in the following the coarsening factor will be denoted by g. The block smoother uses small cubic blocks with side length equal to the coarsening ratio.

The multigrid method itself uses a simple data layout with lexicographic ordering of the unknowns, the numbering of the grid points includes the boundary values, c.f., Fig. 1, which depicts the two-dimensional analogue of the distribution scheme used. This numbering is neglecting the blocking and it is used

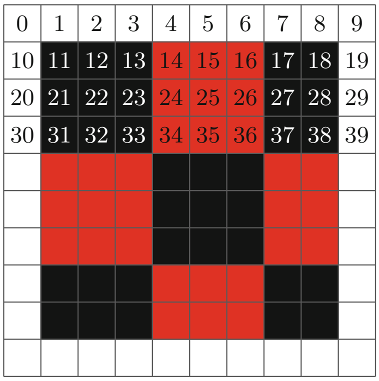

Fig. 1. Numbering of the grid points for the multigrid method

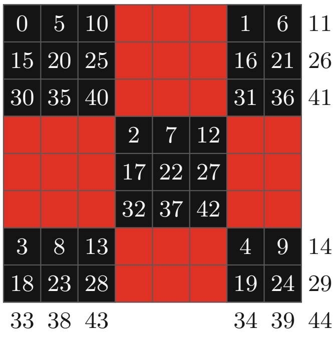

Fig. 2. Numbering of the grid points for the block smoother

for all operations, i.e., calculating the residual, restriction of the residual, and prolongation of the error and correcting the current approximation afterwards.

In order for the block smoother to benefit from the spatial locality of the unknowns of a block the data can be rearranged in memory such that data belonging to one block is stored consecutively in memory. Further, for data access to be as fast as possible on a GPU coalesced access has to be used. Therefore the unknowns of 16 blocks of the same color are interleaved to provide coalesced data access for one half warp. A 2D example for a half warp size of 5 is depicted in Fig. 2. Similar approaches have been used, e.g., in [13].

As the smoother does not need boundary values, the boundaries are not included in this data layout. To accommodate for missing values in subdomains at the boundary padding is used.

The usage of two data layouts results in memory copies before and after a relaxation. Overall we obtain Algorithm 2 for a V-cycle.

We compared this algorithm to an alternative using the previously defined simplified layout, only. To obtain the highest possible performance instead of a red and a black block-sweep two full block Jacobi-sweeps are performed, where the blocks are inverted as in the previous algorithm by a few iterations of Gauß–Seidel. The resulting V-cycle is given by Algorithm 3.

Algorithm 2. V-cycle with block smoother and two data layouts

```
for $\ell = 1, \ldots, \ell_{\max} - 1$ do
  for $k = 1, \ldots, \nu_1$ do
    $r_\ell = f_\ell - L_\ell u_\ell$
    Copy residual from standard layout to layout for block smoother
    Solve $L_\ell e_\ell = r_\ell$ on red blocks using $\sigma$ iterations of Gauß-Seidel
    Copy residual from layout for block smoother to standard layout
    $u_\ell = u_\ell + e_\ell$
    $r_\ell = f_\ell - L_\ell u_\ell$
    Copy residual from standard layout to layout for block smoother
    Solve $L_\ell e_\ell = r_\ell$ on black blocks using $\sigma$ iterations of Gauß-Seidel
    Copy residual from layout for block smoother to standard layout
    $u_\ell = u_\ell + e_\ell$
  end for
  $f_{\ell+1} = R_\ell(f_\ell - L_\ell u_\ell)$
end for
$u_{\ell_{\max}} = L_{\ell_{\max}}^{-1} f_{\ell_{\max}}$
for $\ell = \ell_{\max} - 1, \ldots, 1$ do
  $e_\ell = P_\ell u_{\ell+1}$
  $u_\ell = u_\ell + e_\ell$
  for $k = 1, \ldots, \nu_2$ do
    Copy residual from standard layout to layout for block smoother
    Solve $L_\ell e_\ell = r_\ell$ on red blocks using $\sigma$ iterations of Gauß-Seidel
    Copy residual from layout for block smoother to standard layout
    $u_\ell = u_\ell + e_\ell$
    $r_\ell = f_\ell - L_\ell u_\ell$
    Copy residual from standard layout to layout for block smoother
    Solve $L_\ell e_\ell = r_\ell$ on black blocks using $\sigma$ iterations of Gauß-Seidel
    Copy residual from layout for block smoother to standard layout
    $u_\ell = u_\ell + e_\ell$
  end for
end for
```

To stop the iteration the 2-norm of the residual is checked after each V-cycle. The calculation of this 2-norm is carried out by squaring all entries and then using a fan-in scheme using two arrays that are used alternately to sum up two variables in one array and storing them in the other until the sum of all squared entries is located in the first entry of one array. This allows to use the GPU for this task, as well.

5 Numerical Results

As test problem we consider the PDE

$$\begin{aligned} \Delta u(x) &= f(x), \qquad \text{for} x \in \Omega = (0,1)^3, \\ u(x) &= 0, \qquad \text{for} x \in \partial\Omega. \end{aligned}$$

Algorithm 3. V-cycle with block smoother one data layout

```
for $\ell = 1, \ldots, \ell_{\max} - 1$ do
    for $k = 1, \ldots, 2\nu_1$ do
        $r_\ell = f_\ell - L_\ell u_\ell$
        Solve $L_\ell e_\ell = r_\ell$ on all blocks using $\sigma$ iterations of Gauß-Seidel
        $u_\ell = u_\ell + e_\ell$
    end for
    $f_{\ell+1} = R_\ell(f_\ell - L_\ell u_\ell)$
end for
$u_{\ell_{\max}} = L_{\ell_{\max}}^{-1} f_{\ell_{\max}}$
for $\ell = \ell_{\max} - 1, \ldots, 1$ do
    $e_\ell = P_\ell u_{\ell+1}$
    $u_\ell = u_\ell + e_\ell$
    for $k = 1, \ldots, 2\nu_2$ do
        Solve $L_\ell e_\ell = r_\ell$ on all blocks using $\sigma$ iterations of Gauß-Seidel
        $u_\ell = u_\ell + e_\ell$
    end for
end for
```

The right hand side f was chosen as $3\pi^2 \sin(\pi x_1) \sin(\pi x_2) \sin(\pi x_3)$ such that the analytical solution of the problem is given by

$$u(x) = \sin(\pi x_1) \sin(\pi x_2) \sin(\pi x_3).$$

The problem was discretized using 7-point finite differences.

The implementation was tested in single precision on a NVIDIA Tesla M2050 GPU in the JuDGE cluster at the Jülich Supercomputing Centre. Single precision is sufficient for smaller problems, only, but it can be used as an efficient preconditioner even in the double precision case [15]. The M2050 GPU provides a theoretical peak performance of 1.03 TFLOPS in single precision. In any case, the block size and the coarsening ration were chosen to be the same, resulting in less coarse grids when larger block sizes were chosen.

First, we compare the time for one V-cycle and the performance achieved for Algorithms 2 and 3. The results can be found in Table 1. As expected the necessary copying of the data corrupts the performance a lot, even though Algorithm 3 does twice the amount of operations, the time needed for a V-cycle is smaller.

As expected from a theoretical point of view both algorithms do behave similarly regarding the convergence rate. A plot of the convergence history of both methods can be found in Fig. 3.

As the performance of Algorithm 3 using one data layout, only, was superior, we measured the time to solution with this algorithm, only. We measured the time needed to reduce the error to the discretization error and calculated the obtained performance. In each case in the block Jacobi method the systems belonging to one block were solved approximately with 10 iterations of Gauß–Seidel. The result can be found in Table 2.

Obviously, a block smoother results in a much better performance in terms of GFLOPS when a large block size is chosen. On the other hand, as the coarsening

Table 1. Performance of Algorithms 2 and 3 for $2^5 + 1$, $2^6 + 1$, and $2^7 + 1$ grid points in each direction.

n	$2^5 + 1 = 33$		$2^6 + 1 = 65$		$2^7 + 1 = 129$	
	Time	Performance	Time	Performance	Time	Performance
1 data layout	0.169 s	4.41 GFLOPS	0.302 s	19.76 GFLOPS	1.211 s	39.43 GFLOPS
2 data layouts	0.218 s	1.90 GFLOPS	0.405 s	8.09 GFLOPS	1.263 s	20.72 GFLOPS

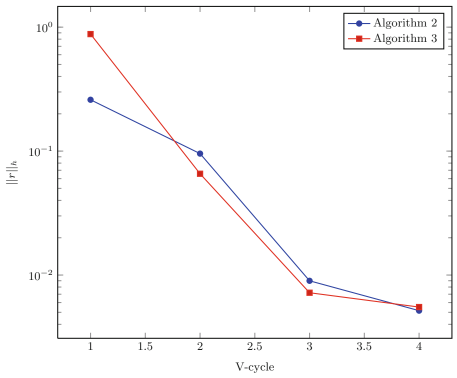

Fig. 3. Convergence history using $6^3 + 1$ grid points in each direction.

Table 2. Time to solution and achieved performance of Algorithm 3 for different grid sizes and coarsening ratios.

n	Block size	Time/iter	#iterations	Performance
$2^5 - 1$	2×2	0.156 s	3	4.76 GFLOPS
$2^6 - 1$	2×2	0.322 s	3	18.50 GFLOPS
$2^7 - 1$	2×2	1.129 s	3	42.25 GFLOPS
$3^5 - 1$	3×3	4.222 s	3	70.28 GFLOPS
$4^3 - 1$	4×4	0.211 s	4	25.11 GFLOPS
$4^4 - 1$	4×4	4.765 s	4	71.22 GFLOPS
$6^3 - 1$	6×6	2.673 s	5	75.43 GFLOPS

ratio is increased as well, the performance in terms of necessary iterations is worsening. Overall, the time to solution is reduced nevertheless, c.f., the time needed for the solution of a system with 63^3 unknowns: When a block size of 2 is used, 3 iterations are needed and each iteration takes 0.322 s, but when a block size of 4 is used, we need one iteration more but each iteration now only takes 0.211 s. The time to solution is 0.966 s in the first case and 0.844 s in the second, so the second approach only takes 87 % of the time of the first.

6 Conclusion and Outlook

Block smoothers provide a way to increase the amount of local arithmetic operations in a way beneficial for multigrid methods. The smoothers allow for a more aggressive coarsening resulting in less coarse grids while at the same time exploiting modern computer architectures. The inclusion of a special data layout for the smoothers, only, does not result in a higher performance, but even a simplistic straightforward approach yields a reduction in time to solution.

We are currently working on analyzing block smoothers theoretically and in incorporating the ideas exploited here in a parallel multigrid method targeting massively parallel computers.

Acknowledgments. We acknowledge the Jülich Supercomputing Centre for providing access to the JuDGE cluster.

References

1. Bakhvalov, N.S.: On the convergence of a relaxation method with natural constraints on the elliptic operator. USSR Comp. Math. Math. Phys. **6**, 101–135 (1966)
2. Bolten, M., Kahl, K.: Using block smoothers in multigrid methods. PAMM **12**, 645–646 (2012)
3. Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comp. **31**(138), 333–390 (1977)
4. Briggs, W.L., Henson, V.E., McCormick, S.F.: A Multigrid Tutorial. SIAM, Philadelphia (2000)
5. Chow, E., Falgout, R.D., Hu, J.J., Tuminaro, R.S., Yang, U.M.: A survey of parallelization techniques for multigrid solvers. In: Heroux, M.A., Raghavan, P., Simon, H.D. (eds.) Parallel Processing for Scientific Computing, chap. 10, SIAM Series on Software, Environments, and Tools, SIAM, Philadelphia (2006)
6. Fedorenko, R.P.: A relaxation method for solving elliptic difference equations. USSR Comp. Math. Math. Phys. **1**(5), 1092–1096 (1962)
7. Fedorenko, R.P.: The speed of convergence of one iterative process. USSR Comp. Math. Math. Phys. **4**(3), 227–235 (1964)
8. Göddeke, D., Strzodka, R.: Mixed precision GPU-multigrid solvers with strong smoothers. In: Kurzak, J., Bader, D.A., Dongarra, J.J. (eds.) Scientific Computing with Multicore and Accelerators, chap. 7, CRC Press, December 2010
9. Goddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Wobker, H., Becker, C., Turek, S.: Using gpus to improve multigrid solver performance on a cluster. Int. J. Comput. Sci. Eng. **4**(1), 36–55 (2008)
10. Haase, G., Liebmann, M., Douglas, C.C., Plank, G.: A parallel algebraic multigrid solver on graphics processing units. In: Zhang, W., Chen, Z., Douglas, C.C., Tong, W. (eds.) HPCA 2009. LNCS, vol. 5938, pp. 38–47. Springer, Heidelberg (2010)
11. Hackbusch, W.: On the convergence of a multi-grid iteration applied to finite element equations. Report 77–8, Institute for Applied Mathematics, University of Cologne, West Germany, Cologne (1977)
12. Hackbusch, W.: On the multi-grid method applied to difference equations. Computing **20**, 291–306 (1978)

13. Müthing, S., Ribbrock, D., Göddeke, D.: Integrating multi-threading and accelerators into DUNE-ISTL. In: Abdulle, A., Deparis, S., Kressner, D., Nobile, F., Picasso, M. (eds.) Numerical Mathematics and Advanced Applications - ENUMATH 2013. Lecture Notes in Computational Science and Engineering, vol. 103, pp. 601–609. Springer International Publishing, Switzerland (2015)
14. Smith, B., Bjorstad, P., Gropp, W.: Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, Cambridge (2004)
15. Tadano, H., Sakurai, T.: On single precision preconditioners for Krylov subspace iterative methods. In: Lirkov, I., Margenov, S., Waśniewski, J. (eds.) LSSC 2007. LNCS, vol. 4818, pp. 721–728. Springer, Heidelberg (2008)
16. Trottenberg, U., Oosterlee, C., Schüller, A.: Multigrid. Academic Press, San Diego (2001)