Scalable Multi-coloring Preconditioning for Multi-core CPUs and GPUs

Vincent Heuveline[1], Dimitar Lukarski[1,2], and Jan-Philipp Weiss[1,2]

[1] Engineering Mathematics and Computing Lab (EMCL)
[2] SRG New Frontiers in High Performance Computing,
Karlsruhe Institute of Technology, Germany
{vincent.heuveline,dimitar.lukarski,jan-philipp.weiss}@kit.edu

Abstract. Krylov space methods like conjugate gradient and GMRES are efficient and parallelizable approaches for solving huge and sparse linear systems of equations. But as condition numbers are increasing polynomially with problem size sophisticated preconditioning techniques are essential building blocks. However, many preconditioning approaches like Gauss-Seidel/SSOR and ILU are based on sequential algorithms. Introducing parallelism for preconditioners is mostly hampering mathematical efficiency. In the era of multi-core and many-core processors like GPUs there is a strong need for scalable and fine-grained parallel preconditioning approaches. In the framework of the multi-platform capable finite element package HiFlow[3] we are investigating multi-coloring techniques for block Gauss-Seidel type preconditioners. Our approach proves efficiency and scalability across hybrid multi-core and GPU platforms.

Keywords: Parallel preconditioners, multi-coloring, Gauss-Seidel, multi-core CPU, GPU, performance analysis.

1 Introduction

Solution methods for linear systems of equations fall into direct methods like LU decomposition or FFT, and iterative methods like splitting methods (Jacobi, Gauss-Seidel, SSOR), Krylov space methods (CG, GMRES) or multigrid solvers. For iterative methods the number of iterations for reaching a prescribed error tolerance depends on the structure of the iteration matrix, in particular on its eigenvalues and condition number. Preconditioning techniques are used to influence the structure and the spectrum of the matrix. On the one hand, the number of necessary iterations shall be reduced, on the other hand extra work for solving additional linear systems shall not outweigh associated benefits.

In the era of multi-core and many-core computing particular emphasis has to be put on fine-grained parallelism within preconditioning approaches. In this work we consider node-level preconditioning techniques on hybrid multi-core CPU and GPU platforms. The impressive power of GPUs originates from processing thousands of lightweighted threads on huge arrays performing uniform operations and coalesced memory transfers via highly capable on-device data paths. For many applications the limitations of the PCIe connection to the host

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 389–397, 2011.

machine are considerable bottlenecks. Moreover, programmability is a major challenge for hybrid and heterogeneous computing platforms since extensions by accelerators introduce different processing models and programming interfaces.

The parallel HiFlow^3 finite element package [8] is built on a two-level library with an inter-node level communication layer based on MPI and an intra-node communication and computation model: the local multi-platform linear algebra toolbox (lmpLAtoolbox) [7]. By unified interfaces with backends to different platforms and accelerators it allows seamless integration of various numerical libraries and devices. The user is freed from any particular hardware knowledge – the final decision on platform and chosen implementation is taken at run time.

This paper investigates parallel symmetric block Gauss-Seidel type preconditioning based on multi-coloring techniques for GPU and CPU platforms. It evaluates performance and scalability characteristics and shows performance benefits for three matrix systems on diverse hybrid multi-core CPU and GPU platforms. Our test scenarios and environments unveil particular behavior with respect to core and memory configuration. Our approach provides an out-of-the box type node-level preconditioning approach for general purpose utilization as well as for use within the complex finite element package HiFlow^3 [8].

To our knowledge there is not much work about parallel preconditioning techniques on GPUs in the literature – although it is a highly important topic. In [5] fined-grained parallel preconditioners and multigrid smoothers are considered for GPUs. But, the approach is limited to banded matrices based on linewise numbering of generalized tensor product meshes. Chebyshev type preconditioners [1] have the appealing of fine-grained parallelism within basic linear algebra routines. However, detailed knowledge of the spectrum of the matrix is required and hence, it is not applicable as a general purpose parallel technique.

2 Parallel Preconditioning Techniques

Due to their work complexity splitting methods are a non-optimal choice for the solution of huge linear system. However, they play an important role as preconditioners for Krylov space methods and smoothers for multigrid solvers. For the Poisson model problem on regular grids Jacobi and Gauss-Seidel methods have an asymptotic complexity of $O(N^{1+2/d})$ where N is the number of unknowns and $d = 1, 2, 3$ is the problem dimension. For Krylov space methods like the conjugate gradient (CG) method work complexity for solving the model problem is only $O(N^{1+1/d})$. By choosing the optimal relaxation parameter SSOR preconditioning for CG reduces work complexity to $O(N^{1+1/(2d)})$ [2]. This is still not the optimal order $O(N)$ like for multigrid methods but this approach circumvents complex treatment of mesh hierarchies, grid transfer operators, and limited parallel scalability due to communication imbalance on coarse grids. An alternative approach is given by algebraic multigrid methods that do not rely on representations of PDEs or any geometric information. Only matrix structures and size of matrix elements are analyzed for constructing a hierarchy of operators. While parallelization of the solution phase is straightforward, this is

not true for the setup phase [6]. Hence, we concentrate on Gauss-Seidel type preconditioning. For our investigation we consider the CG-method for symmetric positive definite matrices. Similar to the ILU-preconditioning approach the relaxed symmetric Gauss-Seidel (SSOR) method is purely sequential. By using the splitting $A = D + L + R$ for the system matrix A with diagonal matrix D, strict lower part L, and strict upper part R the SSOR preconditioning within each CG-step requires solution of the linear system

$$Mz = r \text{ with } M = \frac{1}{\omega(2-\omega)}(D+\omega L)D^{-1}(D+\omega R).$$

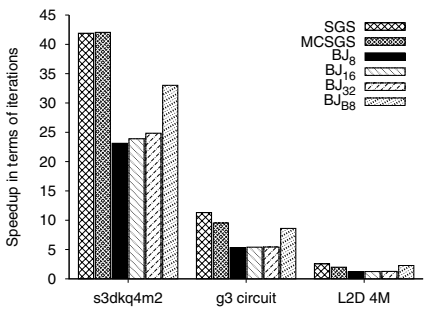
Since $D+\omega L$ and $D+\omega R$ are triangular systems, the solution procedure is typically sequential. For structured grids and stencil operations parallelism can be introduced by red-black or wavefront ordering of nodes which are not applicable to general matrices and for the latter case has varying degree of parallelism. The restriction to simple Jacobi preconditioners with $M = D$ is highly parallel but without positive effect for many linear systems.

An increased level of parallelism can be introduced by block decomposition of the matrix A as shown in Figure 1 (right). The block diagonal matrix D now consists of B blocks $D_1, \ldots, D_B$. Each block row i, $i = 1, \ldots, B$, has $i-1$ left blocks $L_{i,j}$, $j = 1, ..., i-1$ and $B-i$ right blocks $R_{i,j}$, $j = 1, \ldots, B-i$. The block type solution of $Mz = r$ now reads

$$x_i := D_i^{-1}(r_i - \sum_{j=1}^{i-1} L_{i,j}x_j) \text{ for } i = 1, \ldots, B, \tag{1}$$

$$y_i := D_i^{-1}x_i \text{ for } i = 1, \ldots, B, \tag{2}$$

$$z_i := D_i^{-1}(y_i - \sum_{j=1}^{B-i} R_{i,j}z_{i+j}) \text{ for } i = B, \ldots, 1. \tag{3}$$



D_1	R_{11}	R_{12}	R_{13}
L_{21}	D_2	R_{21}	R_{22}
L_{31}	L_{32}	D_3	R_{31}
L_{41}	L_{41}	L_{41}	D_4

Fig. 1. Speedup in terms of the ratio of necessary number of iterations of the unpreconditioned system to the necessary number of iterations of the preconditioned system (left); Example of 4-by-4 block-decomposed matrix (right) for $B = 4$; Problem definitions are given in Table 2

The diagonal blocks D_i, $i = 1, \ldots, B$, are square with size $b_i \times b_i$ but b_i may be different for all i. The vectors x_k, y_k and z_k, $k = 1, \ldots, B$, are block vectors of length b_k. The bracket expressions in the right hand sides of (1) and (3) now consist of $i-1$ and $B-i$ matrix-vector products with vector length b_j and b_{i+j}. In total $B^2 - B$ sparse matrix-vector products and $3B$ sparse matrix-inversions are necessary to compute (1)-(3). For equal block size the degree of parallelism is $b_k = N/B$. The major difficulty in computing (1)-(3) arises from solving for the diagonal blocks D_i which are non-diagonal itself in general.

The basic idea of the multi-coloring approach is to resolve neighbor dependencies by introducing neighborship classes (colors) such that for non-vanishing matrix elements $a_{i,j}$ with $A = (a_{i,j})_{i,j}$ i and j are not members of the same class (color). A straightforward algorithm for determining the colored index sets is

```
for i=1,...,N Set Color(i)=0;
for i=1,...,N Set Color(i)=min(k>0:k!=Color(j) for j ∈ Adj(i));
```

where $\mathrm{Adj}(i) = \{j \neq i | a_{i,j} \neq 0\}$ are the adjacents to node i [10]. By renumbering the mesh nodes by colors diagonal blocks D_i become diagonal itself, B is the number of colors, and b_k is the number of elements for color k. Inversion of D_i then is only a component-wise scaling of the source vector. Due to the data parallelism of the associated routines on vectors there is no load imbalance even for varying block sizes (if the number of elements per block is reasonably large).

In the following we consider three different types of preconditioners: sequential symmetric Gauss-Seidel preconditioner (SGS), parallel block-Jacobi preconditioner with prescribed uniform block size and block-level symmetric Gauss-Seidel preconditioning for approximate inversion of diagonal blocks (BJ), and the parallel multi-coloring symmetric Gauss-Seidel preconditioner (MCSGS). Increasing the number of blocks in the BJ approach increases parallelism, decreases the level of coupling, but also decreases efficiency of the preconditioner. The drawback of the BJ preconditioner is that parallelism is only given by the number of blocks introduced (degree of parallelism is B). Hence, this approach is not scalable with respect to the number of cores.

The importance of preconditioning within the CG-method is presented in Figure 1 (left) by means of three test problem matrices detailed in Table 2. It shows the speedup factor in terms of the ratio of iteration numbers of the non-preconditioned system to the preconditioned system.

3 Implementation Aspects

Our collection of preconditioning routines is part of the HiFlow[3] finite element package [8], a generic, modularized and template-based C++ implementation of fluid dynamic solvers. Under the roof of a MPI-based communication and computation layer the lmpLAtoolbox provides backends to heterogeneous nodes built of various processor types like multi-core CPUs (OpenMP-parallel, Intel MKL, Atlas) and NVIDIA GPUs (CUBLAS and own CUDA SPMV kernels). Currently, an extension by means of an OpenCL interface and vendor-specific

approaches is under construction. The whole module will be released within the framework of the HiFlow[3] project [8]. More information on the structure of the module and its cross-platform portability can be found in [7].

We have tested our three described preconditioning approaches on platforms detailed in Table 1 and for different matrices taken from [9,11] – see Table 2. The CG method is either implemented on the CPU or on the GPU. We use start vector zero, right hand side constant to one, and stop the solver with relative residual less than 10^{-6}. Preconditioning is performed on the same device with exception of SGS which is solely executed on the CPU due to its sequential nature. All computations are performed in double precision. Matrices and all sub-blocks are stored in compressed sparse row format (CSR). This format is our favorite choice for general matrix structures, whereas DIA and ELL format show benefits for particular matrix structures. See [4,3] for SpMV-kernels in different formats for GPUs. Due to the small number of non-zero elements in FEM matrices we use scalar code (one row per thread) versions on the GPU (see [4]) with and without texture caching. Kernel and thread launch times have a considerable impact on overall performance for small sized matrix problems.

MCSGS relies on a preprocessing step in which the matrix graph is analyzed, the matrix and vectors are permuted, and decomposed into blocks. The number of colors depends on the choice of finite elements (Q1, Q2, or others) and mesh properties. For higher degree elements graph connectivity and number of colors increases. 3D problems typically exhibit more graph-connectivity than 2D problems. In the MCSGS approach the amount of data and work per iteration step is independent of the data decomposition into blocks. As no detailed information is available on matrix properties we use the relaxation parameter $\omega = 1$.

Table 1. CPU and GPU system configuration: Pa/Pi = Pageable/Pinned memory, H2D = host-to-device, D2H = device-to-host, 1c/2c/4c/8c = 1/2/4/8 core(s)

Host			Device			
CPU	BW [GB/s]	H2D [GB/s]	GPU	MEM [GB]	BW [GB/s] BT/daxpy/ddot	D2H [GB/s]
2x Intel Xeon 4c (E5450), 8 cores	8c: 6.14 1c: 2.62	Pa: 1.92 Pi: 5.44	Tesla T10 S1070	4x4	71.8/83.1/83.3	Pa: 1.55 Pi: 3.77
1x Intel Core2 2c (6600), 2 cores	2c: 3.28 1c: 3.08	Pa: 1.76 Pi: 2.57	GTX 480	1.5	108.6/135.0/146.7	Pa: 1.38 Pi: 1.82
1x Intel Core i7 4c (920), 4 cores	4c: 12.07 1c: 5.11	Pa: 5.08 Pi: 5.64	GTX 280	1.0	111.5/124.3/94.8	Pa: 2.75 Pi: 5.31

4 Performance Analysis

Our performance analysis is performed on three different GPU-based platforms detailed in Table 1 giving particular information on the system bandwidth. These configurations and associated results clearly show that not only the internal bandwidth of the GPU but also the PCIe bus speed and the utilization of the

bandwidth per core on the CPU need to be considered. Bandwidth values on the GPU are determined by means of the bandwidth test provided by the CUDA SDK (denoted by BT), for the vector update (daxpy), and for the scalar product (ddot). For maximal bandwidth all cores of the CPU need to be active. All our CPU tests are performed on the dual-socket quadcore Xeon system. Although the Tesla S1070 provides four GPUs we only utilize a single GPU in all our tests. Test matrices are listed in Table 2. For the g3 circuit matrix the decomposition into colors is imbalanced ($689390, 789436, 106502, 150$ entries per color) while for both other examples the block distributions have balanced sizes. In general, smaller matrices (like 3dkq4m2) are better suited for the cache-oriented CPUs.

The MCSGS algorithm shows good scaling properties and load balancing on the test platforms as it is based on fine-grained parallelism of the BLAS 1 vector operations and sparse matrix-vector operations in (1)-(3). For a larger number of unknowns parallelism can be better exploited. As the #colors or #blocks B stays constant the #cores may be increased with N asymptotically. Since the algorithm is bandwidth-bound on most platforms it scales with the bandwidth of the system and hence with core configurations and memory organization.

Table 2. Description and properties of test matrices

Name	Description of the problem	#rows	#non-zeros	#colors	#block-SpMV
s3dkq4m2	FEM - Cylindrical shells	90449	4820891	24	552
g3 circuit	Circuit simulation	1585478	7660826	4	12
L2D 4M	FEM - Q1 Laplace 2D	4000000	19992000	2	2

First, we compare the performance of the non-preconditioned CG solver on our test platforms in Figure 2 - 4. For large matrices we observe that the speedup of the GPU version over the OpenMP-parallel CPU version is basically due to the bandwidth difference. The only exception for the GPU is given for the 3dkq4m2 matrix which is so small that the full internal bandwidth of the GPU device cannot be utilized due to the large number of kernel calls over the PCIe. In all cases we find that texture caching on the GPU slightly improves performance.

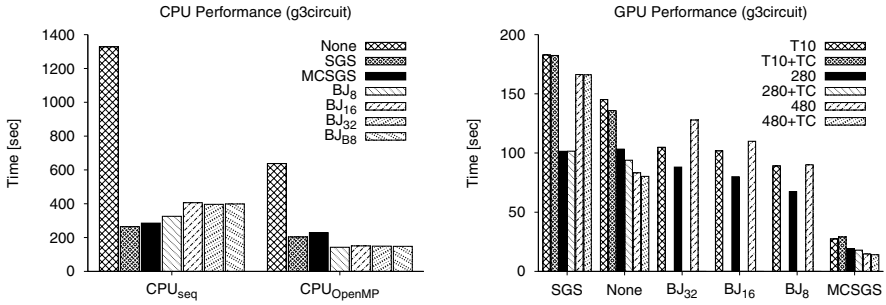


Fig. 2. Performance of preconditioned CG-solver on CPU (OpenMP: 2x4 cores) and GPU (TC: with texture caching) for g3 circuit test matrix

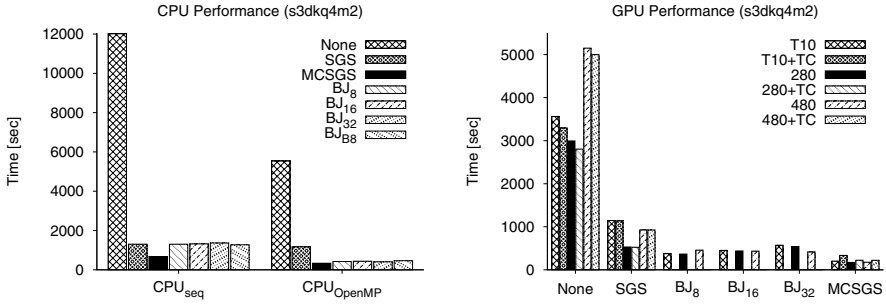


Fig. 3. Performance of preconditioned CG-solver on CPU (OpenMP: 2x4 cores) and GPU (TC: with texture caching) for s3dkq4m2 test matrix

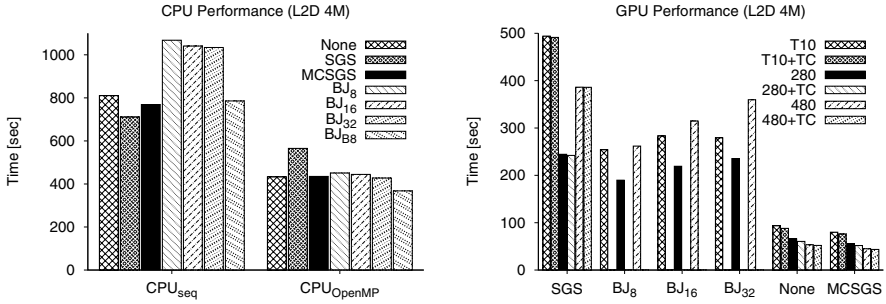


Fig. 4. Performance of preconditioned CG-solver on CPU (OpenMP: 2x4 cores) and GPU (TC: with texture caching) for L2D 4M test matrix

Computation of the SGS preconditioning step sequentially on the CPU – while the CG solver is either performed on the GPU or OpenMP-parallel – emphasizes the limitations due to Amdahl's law.

The BJ preconditioning is improving the number of iterations (see Figure 1) but not in all cases the total time is reduced. The CPU plots in Figures 2-4 show the BJ results in terms of total CG-solver time for block sizes 32, 16, and 8 and for a fixed number of eight blocks (B8). For the L2D 4M matrix on the CPU the best performance is obtained by using BJ preconditioning (see Figure 4) since the CPU cores are optimized for executing large sequential parts.

On the GPU the multi-coloring MCSGS preconditioning yields the best results for all test matrices and CPU-GPU configurations due to its inherent scalable parallelism. Efficiency of the preconditioner is paired with parallel execution within the forward and the backward step. In MCSGS the number of sparse matrix-vector operations in the block decomposition increases quadratically with the number of colors. Therefore, we observe a significant latency for launching GPU kernels and forking CPU threads as the number of colors increases. But even in the scenario with 552 SpMV operations per preconditioning step (s3dq4m2) the speedup over the CPU version is more than a factor of two. For

small matrices with large number of colors the latency is further increased when texture caching is used. Therefore, MCSGS is faster without texture caching on the GPU for the s3dkq4m2 matrix. If the number of unknowns in each color is small and evenly distributed then even for the single-threaded CPU-case a significant speedup of MCSGS is observed (see Figure 3). For the L2D 4M matrix SGS and BJ preconditioning give a breakdown in performance on the GPU. Only the MCSGS preconditioner yields slightly improved results on the GPU.

5 Conclusion

We have tested preconditioning techniques for various matrix problems on hybrid multi-core CPU and GPU systems. Our investigated multi-coloring technique for symmetric Gauss-Seidel preconditioning (MCSGS) provides a scalable and fine-grained parallel approach on the level of basic linear routines. It is a robust solution and applicable to a large class of problems. Hence it can be used as an out-of-the-box approach for any FEM matrix. In our test scenarios MCSGS has delivered best results on both the 2-socket quadcore CPUs and on GPUs. For the s3dkq4m2 matrix MCSGS preconditioning is speeding up the CG method on our GPU systems up to a factor of 17.6. Assessment of performance has shown that the PCIe connection of GPUs is not limiting performance and scalability for this solution. In our ongoing work we consider an extended set of preconditioners (e.g. Chebyschev, non-sysmmetric cases with GMRES). Furthermore, preconditioning techniques in a cluster of multicore-CPU and GPU nodes with parallelisation across several devices is investigated.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and its collaboration partner Hewlett-Packard.

References

1. Asgasri, A., Tate, J.E.: Implementing the Chebyshev Polynomial Preconditioner for the Iterative Solution of Linear Systems on Massively Parallel Graphics Processors. In: CIGRE Canada Conference on Power Systems, Toronto (2009)
2. Axelsson, O., Barker, V.: Finite element solution of boundary value problems: theory and computation. SIAM, Philadelphia (2001)
3. Baskaran, M.M., Bordawekar, R.: Optimizing Sparse Matrix-Vector Multiplication on GPUs. Technical report, IBM (2009)
4. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC 2009: Proc. of the Conf. on High Perf. Computing Networking, Storage and Analysis, pp. 1–11. ACM, New York (2009)
5. Göddeke, D.: Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Cluster. PhD thesis, Technische Universität Dortmund (2010)

6. Van Henson, E., Yang, U.M.: BoomerAMG: a parallel algebraic multigrid solver and preconditioner. Appl. Numer. Math. 41(1), 155–177 (2002)
7. Heuveline, V., Subramanian, C., Lukarski, D., Weiss, J.P.: A multi-platform linear algebra toolbox for finite element solvers on heterogeneous clusters. In: PPAAC 2010, IEEE Cluster 2010 Workshops (2010)
8. HiFlow[3], http://www.hiflow3.org
9. Information Technology Laboratory of the National Institute of Standards and Technology, http://math.nist.gov/MatrixMarket/
10. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia (2003)
11. University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices/