Sparse Approximate Inverse Preconditioners for Iterative Solvers on GPUs

Mykola Lukash[1], Karl Rupp[1,2], Siegfried Selberherr[1]
[1] **Institute for Microelectronics, TU Wien**
Gußhausstraße 27-29/E360, 1040 Wien, Austria
[2] **Institute for Analysis and Scientific Computing, TU Wien**
Wiedner Hauptstraße 8-10/E101, 1040 Wien, Austria
nlukash@gmail.com, {rupp|selberherr}@iue.tuwien.ac.at

Abstract

For the solution of large systems of linear equations, iterative solvers with preconditioners are typically employed. However, the design of preconditioners for the black-box case, in which no additional information about the underlying problem is known, is very difficult. The most commonly employed method of incomplete LU factorizations is a serial algorithm and thus not well suited for the massively parallel computing architecture of GPUs. We investigate sparse approximate inverse preconditioners in this work, which show a very high degree of parallelism. The preconditioner setup is accomplished in a hybrid manner, where parts of the algorithm which require dynamic memory allocations are carried out on the CPU, while the GPU is used for the computationally expensive factorizations. Our benchmark results demonstrate that our implementations in ViennaCL are well suited as a black-box preconditioner for multi- and many-core architectures.

1. INTRODUCTION

Discretization schemes for partial differential equations such as the Finite Difference, the Finite Element, or the Finite Volume scheme ultimately lead to the need for the solution of a large system of linear equations. Since the coupling between the equations is usually rather weak, the system matrix is very sparse, allowing for millions of unknowns on average workstations. For such huge systems, iterative solution methods are typically employed, where the convergence rate depends on the condition number of the iteration matrix. Consequently, in practical applications it is of interest to keep the condition number low, for which so-called preconditioners are employed. Formally, one way of interpreting the action of a preconditioner is to multiply the original system

$$Ax = b$$

with a matrix M to get

$$MAx = Mb$$

and to solve the modified system

$$\tilde{A}x = \tilde{b} \ ,$$

where $\tilde{A} = MA$ and $\tilde{b} = Mb$. If M is in a certain sense a good approximation to the inverse of A, the iterative solution of the modified system converges considerably faster.

Many different variants of preconditioners exist. Simple preconditioners may modify A and b directly, but typically they act on the residual $r_k = Ax_k - b$ in the k-th iterate x_k of the solver. Therefore, the preconditioner matrix M is not set up in an explicit form, but rather implicitly defined over the action on the residual.

A lot of effort has been spent on the construction of preconditioners for general linear systems. Among the most popular methods used today are incomplete LU factorizations (ILU) [1], algebraic multigrid preconditioners (AMG) [2], and sparse approximate inverses (SPAI) [3]. While the different flavors of ILU are used for many different classes of matrices, the major downside of the method is its serial nature. Block-ILU variants have been developed as a remedy, but the price to pay is that more iterative solver iterations are necessary.

In recent years, graphics processing units (GPUs) are used not only for graphics applications, but also for certain special purpose computations, e.g. [4]. The most frequently used development platform is CUDA [5] by NVIDIA, which is taylored to products of a single vendor. In contrast, the open standard OpenCL maintained by the Khronos Group [6] provides a unified interface to hardware from different vendors. In particular, it allows to program GPUs as well as CPUs using the same language.

The massively parallel single-instruction-multiple-data (SIMD) architecture of GPUs is well suited for stream processing problems. However, the processing units must be provided with data in order to use them efficiently. This can be achieved by loading data to on-chip memory, thus reducing global memory accesses, and then operate on the fast on-chip cache only. Such caching strategies have been successfully employed for dense linear algebra operations as defined on BLAS level 3 [7]. On the contrary, iterative solvers for sparse systems have a higher emphasis on memory bandwidth due to the use of operations defined on BLAS levels 1 and 2 only.

Our general purpose free open source linear algebra library ViennaCL provides high-level access to various iterative solvers. In addition, it offers simple preconditioners like

a Jacobi preconditioner [1] with full GPU acceleration. The ILU preconditioner provided with ViennaCL is due to its serial nature currently always executed on the CPU, causing a lot of data transfer overhead between CPU and GPU. To remove this bottleneck, this work presents the results obtained from the implementation of two SPAI preconditioners: First, a SPAI implementation as proposed by Grote and Huckle [3] suitable for general matrices is considered, and second a factored SPAI (FSPAI) variant suitable for symmetric positive definite matrices [9] is presented.

2. SPAI

First we recapitulate the most important concepts of SPAI. In particular, the main advantages are as follows:

- SPAI is inherently parallel, because each column (or row respectively) of the preconditioner matrix M can be processed in parallel.
- SPAI can autonomously identify new entries for the sparsity pattern of the preconditioner matrix M.
- The application of SPAI to the current residual in iterative solvers reduces to a matrix-vector product, which can be well parallelized [10, 11].

Therefore, both the setup of the preconditioner as well as the application to the residual allow for a massively parallel execution model.

2.1. Static SPAI - Theoretical Background

The essence of the SPAI algorithm is to determine a sparse matrix M which minimizes

$$\|AM - I\|_{\mathrm{F}}^2 , \qquad (1)$$

where M has a prescribed non-zero pattern, I denotes the identity matrix and $\|\cdot\|_{\mathrm{F}}$ denotes the Frobenius norm. It should also be noted that M recovers the inverse of A, if no restriction on the non-zero pattern of M is imposed.

The parallel nature of SPAI can be readily seen when rewriting (1) as

$$\|AM - I\|_{\mathrm{F}}^2 = \sum_{k=1}^{n} \|(AM-I)e_k\|_2^2 = \sum_{k=1}^{n} \| Am_k - e_k \|_2^2, \quad (2)$$

where m_k denotes the k-th column of M and analogously e_k denotes the k-th column of the identity matrix I. Each summand in (2) represents an independent least-squares problem for one column m_k of M. For the solution of each of the least-squares problems, a QR decomposition is employed.

Typical choices for the sparsity pattern of M are a diagonal pattern or the pattern of A. Inspired by Neumann series and similar to ILU preconditioners, the patterns of A^2 and A^3 can also be used, provided that the pattern of the matrix powers can be computed efficiently. As outlined in the next section, the possibility to dynamically update the non-zero pattern of M makes the choice of the initial pattern less crucial.

For a column m_k of M it is sufficient to consider the index set $\mathcal{J}_k$ for the non-zeros of m_k:

$$\mathcal{J}_k = \left\{ j \in \{1,\ldots,n\} : m_k(j) \neq 0 \right\}$$

We denote the cardinality of $\mathcal{J}_k$ with q.

Similarly, for the matrix-vector product Am_k only the columns of A with indices from the set $\mathcal{J}_k$ may lead to non-zero contributions. We denote with I_k the set of non-zero rows of the submatrix matrix $A(.,\mathcal{J}_k)$:

$$I_k = \left\{ i \in \{1,\ldots,n\} : \sum_{j\in\mathcal{J}} |a_{ij}| \neq 0 \right\} .$$

Consequently, it is sufficient to consider the minimization problem on the condensed sytem $A(I_k,\mathcal{J}_k)$. We introduce the following abbreviations:

$$\hat{A}_k = A(I_k,\mathcal{J}_k) \in \mathbb{R}^{p\times q}$$
$$\hat{m}_k = m_k(\mathcal{J}) \in \mathbb{R}^{q\times 1}$$
$$\hat{e}_k = e_k(I) \in \mathbb{R}^{p\times 1}$$

The least-squares problem can then be written in the compact form

$$\min_{\hat{m}_k} \|\hat{A}\hat{m}_k - \hat{e}_k\|_2, \quad k = 1,\ldots,n$$

Usually, the matrix $\hat{A}$ is dense and in typical matrices obtained from linear finite element discretizations the dimensions of the blocks are in the range $p = 20,\ldots,40$ and $q = 10,\ldots,20$, which is small enough for GPU caches. If A is a regular matrix, $\hat{A}_k$ has full rank q, and we can use QR decomposition for solving the least-squares problems. The non-zero entries of m_k are then computed as

$$\hat{c} = Q^T \hat{e}_k ,$$
$$\hat{m}_k = R(\mathcal{Q},\mathcal{Q})^{-1}\hat{c}(\mathcal{Q}) , \quad \mathcal{Q} = \{1,\ldots,q\}$$

The matrix-vector product with Q^T is carried out implicitly using the chain multiplication of Householder reflections determined by the Householder vectors stored in the lower-triangular part of the matrix R.

2.2. Dynamic SPAI

The ability of dynamically updating the sparsity pattern of the preconditioner matrix M adds considerably to the attractiveness of the method. For a given preconditioner M with

k-th column m_k, the contribution to the residual is found from (2) as

$$r_k = Am_k - e_k.$$

If $r_j = 0, \forall j = 1, \ldots, n$, then clearly $M = A^{-1}$. However, since the inverse of a sparse matrix is typically dense, the residual vector is non-zero in most cases. We denote the set of indices of non-zero entries of r_k by

$$\mathcal{L}_k = \left\{ l : r_l \neq 0 \right\}.$$

For each $l \in \mathcal{L}_k$ the set of column-indices for the non-zero entries in the l-th row of A is

$$\mathcal{N}_l = \left\{ j : a_{lj} \neq 0 \right\}.$$

Thus, only indices in $\mathcal{N}_l$ are new candidates for the new non-zero pattern in m_k for the elimination of the respective non-zero value in r_k. The union of all sets $\mathcal{N}_l$ for all $l \in \mathcal{L}_k$ is the set of indices that can reduce the Euclidean norm of the residual vector:

$$\tilde{\mathcal{J}} = \bigcup_{l \in \mathcal{L}} \mathcal{N}_l.$$

Since the residual norm $\|r_k\|_2$ must be reduced, we note that new entries are obtained by a solution of the one-dimensional minimization problem:

$$\min_{\mu_j} \|A(m_k + \mu_j e_j) - e_k\| = \min_{\mu_j} \|r_k + \mu_j A e_j\|_2$$

Denoting the j-th column of A with a_j, there holds $Ae_j = a_j$ and we obtain the term

$$\min_{\mu_j} \|r_k + \mu_j a_j\|_2^2 \Leftrightarrow \|r_k\|_2^2 + 2\mu_j \langle r_k, a_j \rangle + \mu_j^2 \|a_j\|_2^2 = 0.$$

The minimizer is

$$\mu_j = -\frac{\langle r_k, a_j \rangle}{\|a_j\|_2^2}.$$

For each of the potential new index s from $\mathcal{N}_j$ to be added to the index set $\mathcal{J}_k$, one consequently computes the tentative new residuals

$$\|r_k\|_2^2 - \frac{\langle r_k, a_j \rangle^2}{\|a_j\|_2^2},$$

for which it suffices to consider

$$\rho_s = \frac{\langle r_k, a_j \rangle^2}{\|a_j\|_2^2}$$

as a criterion for acceptance of the new index. There are several strategies on the selection of the new indices possible: Either one takes only indices with the largest values of ρ_s, or indices are selected based on a specified threshold. As a consequence, the set I is enlarged by $\tilde{I}$, where $\tilde{I}$ is set of non-zero rows in $A(., \mathcal{J} \cup \tilde{\mathcal{J}})$, resulting in the extended Least Squares problem

$$\overline{A} = A(I \cup \tilde{I}, \mathcal{J} \cup \tilde{\mathcal{J}}) \in \mathbb{R}^{(p+\tilde{p}) \times (q+\tilde{q})}$$

where $\tilde{p} = |\tilde{I}|$ and $\tilde{q} = |\tilde{\mathcal{J}}|$. The new matrix $\tilde{A}$ can be factorized as

$$\tilde{A} = \begin{pmatrix} \hat{A} & A(I, \tilde{\mathcal{J}}) \\ 0 & A(\tilde{I}, \tilde{\mathcal{J}}) \end{pmatrix} = \begin{pmatrix} Q & 0 \\ 0 & I_{\tilde{p}} \end{pmatrix} \begin{pmatrix} R & Q_1{}^T A(I, \tilde{\mathcal{J}}) \\ 0 & Q_2{}^T A(I, \tilde{\mathcal{J}}) \\ 0 & A(\tilde{I}, \tilde{\mathcal{J}}) \end{pmatrix}$$
$$= \begin{pmatrix} Q & 0 \\ 0 & I_{\tilde{p}} \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & B_2 \end{pmatrix},$$

which allows for a reuse of the compuations from the previous QR factorization step. Only a QR factorization of the new block $B_2 \in \mathbb{R}^{p+\tilde{p}-q \times \tilde{q}}$ is required. In order to keep the size of B_2 and thus the computational effort under control, a practical guideline is to limit the number of new entries by the number of non-zero entries in the initial column m_k.

2.3. SPAI - Implementation

Since the action of the SPAI preconditioner on the residual is a sparse matrix-vector multiplication, which has already been studied extensively [10, 11], it is sufficient to focus the discussion of implementation details on the preconditioner setup phase. Due to the high degree of parallelism of SPAI, every column of M can be computed independently during the setup. Therefore, purely CPU-based implementations of SPAI are easily obtained using compiler-based approaches such as OpenMP [12] or library-based threading-approaches such as the Boost.thread library [13]. A direct implementation of SPAI on GPUs, however, is hampered by the observation that the cardinality of the index sets I_k and $\mathcal{J}_k$ cannot be obtained a-priori. Since dynamic memory allocation on GPUs using OpenCL is not possible by now, expensive scans for the required memory would be necessary. Therefore, the index sets I_k and $\mathcal{J}_k$ are set up on the CPU and then copied to GPU. After that, the matrices $A(I_k, \mathcal{J}_k)$ are set up.

The massively parallel architecture of GPUs suggests to solve all Least Squares problems simultaneously. However, overall memory requirements induced by the Least Squares problems may soon exceed the limited GPU RAM, thus only a subset can be processed at the same time. In our implementation the overall memory requirements on the GPU are computed from the size of the index sets I_k and $\mathcal{J}_k$. If GPU RAM turns out to be too small, the work load is split into two chunks of equal size, which are processed one after another on the GPU. This strategy is then applied in a recursive manner to each of the two chunks.

Due to the high degree of parallelism still present in each of the chunks, the full Least Squares problem including the QR factorizations of the blocks $A(I_k, \mathcal{J}_k)$ is carried out on the GPU. For the better utilization of the SIMD architecture of GPUs, one thread per column of each block is used, so that each work group factors one SPAI block. After that, results are copied back to CPU RAM and the residual vector is computed. If the Euclidean norm of the residual is higher than a prescribed tolerance, sparsity pattern updates are carried out as described in the previous section.

To summarize, our implementation of SPAI for a given initial sparsity pattern is as follows:

1. Determine the index sets I_k and $\mathcal{J}_k$ for each $k = 1, \ldots, n$.
2. Compute the memory consumption of the n Least Squares problems and split the work load into chunks.
3. For each chunk, assemble the matrices $\hat{A}_k$ and compute the solution of the Least Squares problems using QR factorization.
4. Copy the results back to CPU RAM and compute the residuals.
5. If further pattern updates are required, compute the augmented index sets $\tilde{I}$ and $\tilde{\mathcal{J}}$, otherwise go to 8.
6. Assemble the new entries in the augmented Least Squares matrix and compute the solution reusing previous QR factorizations.
7. Go back to 4.
8. Write all entries computed in the chunk to M.

3. FSPAI

SPAI only requires regularity of the input matrix A. However, there is a price to pay for this universality: First, due to the independent computation of the entries in the preconditioner matrix M, it may happen that M does not have full rank. While this is unlikely to happen in practise because of round-off errors, it may still lead to poor convergence rates of iterative solvers. Moreover, if the system matrix A is symmetric, M will be non-symmetric in general, hence more expensive iterative solvers for non-symmetric matrices have to be employed.

In many cases the system matrix A is symmetric and positive definite, for which the conjugate gradient algorithm [14] can be used as an iterative solver. Clearly, it is desireable to preserve symmetry and positive definiteness of the system when using a preconditioner. For this purpose, a variant of SPAI based on the Cholesky factorization $A = L_\mathrm{A} L_\mathrm{A}^T$ of the system matrix has been developed [15], which is commonly referred to as factored sparse approximate inverse (FSPAI) preconditioner.

3.1. Static FSPAI

Similar to SPAI, we first consider FSPAI for a static non-zero pattern of the preconditioner. In this section we focus on the description of FSPAI for a given pattern. Again, typical initial patterns for M like that of A, A^2 or A^3 often provide good results even without additional pattern update step.

For a symmetric positive definite system matrix A there exists the Cholesky factorization $A = L_\mathrm{A} L_\mathrm{A}^T$, where L_A is the unknown Cholesky factor. Consequently, we seek for a matrix M such that

$$M = LL^T \approx A^{-1} \leftrightsquigarrow L \approx L_A^{-1}$$

L can be obtained via Frobenius norm minimization [9]

$$\min_L \|L_\mathrm{A} L - I\|_F$$

and then normalized such that $\mathrm{diag}(L^T A L) = I$. As in SPAI, this initially sparse problem can be reduced to multiple dense problems. In addition, the non-zero pattern of L is restricted to the lower triangular part. As suggested in [16], the Frobenius norm minimization is replaced by the minimization of the Kaporin functional

$$\min_L \frac{\frac{1}{n}\mathrm{trace}(L^T A L)}{\det(L^T A L)^{\frac{1}{n}}}, \tag{3}$$

which allows for a more compact form of the minimization algorithm.

Let us denote the k-th column of L by l_k, and the allowed non-zero pattern in l_k by $\mathcal{J}_k$, with $\tilde{\mathcal{J}}_k = \mathcal{J}_k \setminus \{k\}$. Then

$$l_k(\tilde{\mathcal{J}}_k) = -l_{kk} A(\tilde{\mathcal{J}}_k, \tilde{\mathcal{J}}_k)^{-1} A(\tilde{\mathcal{J}}_k, k) \ ,$$

with

$$l_{kk}^2 = \frac{1}{A(k,k) - A(\tilde{\mathcal{J}}_k, k)^T A(\tilde{\mathcal{J}}_k, \tilde{\mathcal{J}}_k)^{-1} A(\tilde{\mathcal{J}}_k, k)}.$$

Consequently, the algorithm for the computation of the k-th column of L can be compactly written as:

$$y_k = A(\tilde{\mathcal{J}}_k, \tilde{\mathcal{J}}_k)^{-1} A(\tilde{\mathcal{J}}_k, k) \tag{4}$$

$$l_{kk} = \frac{1}{\sqrt{A(k,k) - A(\tilde{\mathcal{J}}_k, k)^T y_k}} \tag{5}$$

$$l_k(\tilde{\mathcal{J}}_k) = -l_{kk} y_k \tag{6}$$

As for SPAI, the columns of L can be computed in parallel.

The linear system (4) is solved by a Cholesky factorization of $A(\tilde{\mathcal{J}}_k, \tilde{\mathcal{J}}_k)$. Similar to the QR-factorization for the SPAI algorithm, the Cholesky decomposition is the computationally expensive part of FSPAI, since the complexity rises with the third power of the block matrix sizes.

3.2. Dynamic FSPAI

In analogy to SPAI, an automatic non-zero pattern update is available for FSPAI [15]. Again, the key is a one-dimensional minimization problem for the update of the Cholesky factor. Minimization of the updated Kaporin functional

$$\min \frac{\frac{1}{n}\mathrm{trace}((L^T+\lambda_j e_k e_j^T)A(L+\lambda_j e_j e_k^T))}{\det((L^T+\lambda_j e_k e_j^T)A(L+\lambda_j e_j e_k^T))^{\frac{1}{n}}}$$

with respect to the Cholesky factor perturbed in the k-th column with $l_k + \lambda_j e_j$ leads to

$$\lambda_j = -\frac{A(j,\mathcal{J}_k)l_k(\mathcal{J}_k)}{A(j,j)} \ .$$

The difference to the original Kaporin functional (3) is

$$\frac{\frac{1}{n}}{\det(A)^{\frac{1}{n}}(L(1,1)\cdots L(n,n))^{\frac{2}{n}}} \cdot \tau_j \ ,$$

hence the main criterion for adding new entries to the sparsity pattern is the factor

$$\tau_j = \frac{(A(j,\mathcal{J}_k)l_k(\mathcal{J}_k))^2}{A(j,j)} \ .$$

The larger the value τ_j is, the more the Kaporin functional is reduced, if the j-th entry will be added to the non-zero pattern of $\mathcal{J}_k$.

Similar to pattern updates with SPAI, several update strategies are possible [15]. Again, a typical choice is to restrict the maximum number of new indices per update to the initial number of non-zero entries in the respective column of L. Let $\tilde{\mathcal{J}}_1$ denote a given index set, and $\tilde{\mathcal{J}}_2$ the updated index set ordered such that the first elements in $\tilde{\mathcal{J}}_2$ are given by the entries in $\tilde{\mathcal{J}}_1$. Writing $A_1 := A(\tilde{\mathcal{J}}_1,\tilde{\mathcal{J}}_1)$ and $A_2 := A(\tilde{\mathcal{J}}_2,\tilde{\mathcal{J}}_2)$, the Cholesky factorization of A_2 can be obtained from that of A_1 as

$$\begin{aligned} A_2 &= \begin{pmatrix} A_1 & B \\ B_T & C \end{pmatrix} = \begin{pmatrix} L_1 & 0 \\ U & L_2 \end{pmatrix} \begin{pmatrix} L_1^T & U^T \\ 0 & L_2^T \end{pmatrix} \\ &= \begin{pmatrix} L_1 L_1^T & L_1 U^T \\ U L_1^T & L_2 L_2^T + U U^T \end{pmatrix} \ , \end{aligned}$$

where the matrices B and C arise due to the augmented index $\tilde{\mathcal{J}}_2$. Now, U can be obtained from the equation $B = L_1 U^T$. Then, since

$$\begin{aligned} & C = UU^T + L_2 L_2^T \\ \Leftrightarrow \quad & L_2 L_2^T = C - UU^T \ , \end{aligned}$$

it is sufficient to compute the Cholesky factorization of the matrix $C - UU^T$ only. Therefore, the computational effort increases only moderately with the total number of entries compared to computing a full Cholesky factorization at each update step.

3.3. FSPAI - Implementation

The implementation of FSPAI is similar to SPAI. In particular, the computation of the index sets $\mathcal{J}_k$ is carried out on the CPU in parallel using OpenMP. Then the index sets $\mathcal{J}_k$ are copied to GPU RAM and the blocks $A(\tilde{\mathcal{J}}_k,\tilde{\mathcal{J}}_k)$ are set up. A Cholesky factorization of each block on the GPU using OpenCL is computed for all blocks using one thread per column of each block. Again, the typical sizes of the block matrices allow to compute the factorization in GPU cache. After that, the entries of L are obtained according to (4)-(6).

4. BENCHMARK RESULTS

Our implementation of SPAI and FSPAI are compared with the purely CPU-based ILU preconditioner with threshold (ILUT) included in ViennaCL 1.1.2. The tests are carried out on a machine equipped with an Intel Core 2 Quad Q9550 and a NVIDIA GTX 580 GPU running a 64-bit Funtoo Linux. For better comparison, all preconditioners are used within a BiCGStab [17] solver, even though FSPAI would be used with a conjugate gradient (CG) solver in practice. Solver execution times for FSPAI within CG are typically by a factor of roughly two smaller than for BiCGStab, while setup times and the number of solver iterations are unchanged.

The iterative solvers as well as the preconditioners use the same unified solver interface of ViennaCL. For example, the pure BiCGStab for a system matrix A, a load vector b and a result vector x is called by

```
x = solve(A, b, bicgstab_tag());
```

Similarly, the preconditioners with default parameters are employed using

```
ilut_precond<MatType> ilut(A, ilut_tag());
x = solve(A, b, bicgstab_tag(), ilut);

spai_precond<MatType> spai(A, spai_tag());
x = solve(A, b, bicgstab_tag(), spai);

fspai_precond<MatType> fspai(A, fspai_tag());
x = solve(A, b, bicgstab_tag(), fspai);
```

where `MatType` denotes a compatible generic matrix type (currently from ViennaCL or Boost.UBLAS). In particular, this allows for a fair comparison of the individual benchmarks results, since the same solver implementations for purely CPU-based as well as GPU-based types is used.

As a test case, we consider a system matrix arising from the discretization of a time-dependent convection-diffusion equation in three dimensions using linear finite elements and a backward Euler scheme. Each of the 24 202 rows and columns of the system matrix consists of 14 entries on average. No dynamic updates of the preconditioner are employed, because the initial pattern of the system matrix provides good

	Setup (sec)	Solver (sec)	Memory (MB)
ILUT	1.3	2.9	3.6
SPAI, CPU	5.5	1.5	175
SPAI, GPU	2.9	0.3	175
FSPAI, CPU	0.2	1.6	53
FSPAI, GPU	0.2	0.3	53

Table 1. Comparison of execution times for the preconditioner setup phase, the iterative solver phase and memory footprint of the individual preconditioners.

results. The norms of the relative residuals for a solver without preconditioner, with ILUT, with SPAI and FSPAI using the nonzero pattern of the system matrix are compared in Fig. 1. ILUT requires only 45 iterations to reduce the initial residual by ten orders of magnitude, SPAI and FSPAI require about 100, and the unpreconditioned solver reduces the residual only by three orders of magnitude within the first 100 iterations. Consequently, ILUT is most attractive in terms of the reduction of the number of required solver iterations.

The picture changes completely when looking at the reduction of the residual over the solver execution time, cf. Fig. 2. The residual reduction rate of SPAI and FSPAI on CPU and GPU are by a factor of three and ten higher than for ILUT respectively. Interestingly, the unpreconditioned CPU solver provides a similar residual reduction rate than SPAI and FSPAI for our test case. Consequently, we conclude that it may still pay off to run a high number of iterations with an unpreconditioned solver rather than using a serial preconditioner on GPUs.

Preconditioner setup times are depicted in Tab. 1. FSPAI leads to the shortest setup times, which are by a factor of six smaller than for ILUT. In particular, the setup time of ILUT is larger than the setup and the solver execution time of FSPAI. No further performance gain for the setup of FSPAI on the GPU rather than the CPU could be obtained, mostly because the dynamic memory allocations needed for the final CPU-based assembly of the sparse Cholesky factor and its transpose takes the majority of the execution time for the considered test case. The setup phases of SPAI and FSPAI suffer from a rather high memory footprint, resulting in 50-fold and 15-fold memory requirements compared to ILUT respectively. Nevertheless, in this case no splitting of the overall work load into chunks is necessary.

Comparing preconditioner setup times with solver iteration times in Tab. 1 shows that the use of GPUs puts much more emphasis on the preconditioner setup. Matrix-vector products and the vector operations in the solver phase can benefit much more from GPU acceleration than the algorithms using dynamic memory allocations required in the setup phase. For SPAI, solver cycle times are one quarter of the solver cycle

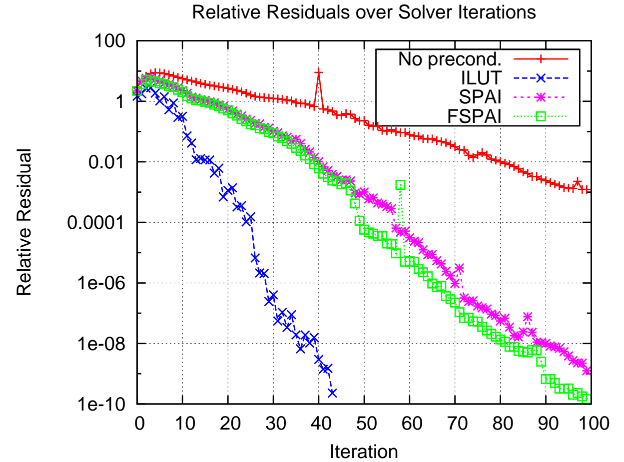

Figure 1. Comparison of relative residual norms over the BiCGStab solver iterations for the different preconditioners. ILUT requires only 45 iterations to reach a relative residual norm of 10^{-10}, while about 100 iterations are required for SPAI and FSPAI respectively.

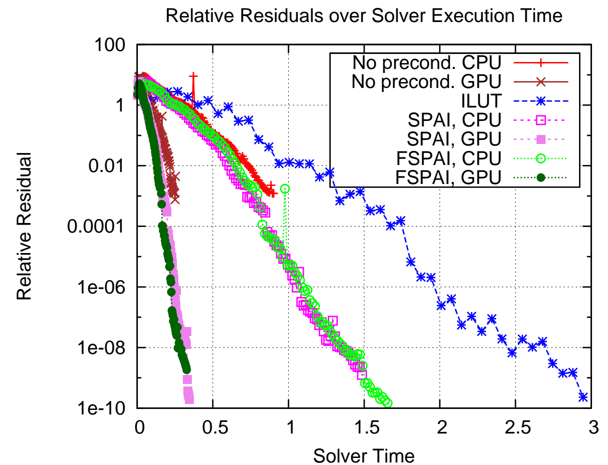

Figure 2. Comparison of the relative residual norms over the BiCGStab execution time. The SPAI and FSPAI preconditioners provide the fastest reduction of the residual with time, while ILUT is clearly a factor of two slower than the CPU-based solvers and a factor of ten slower than the GPU-based solver. Note that the ILUT preconditioner is applied on the CPU even if the solver is executed on the GPU, hence the CPU and the GPU lead to essentially identical timings.

time for the CPU case, while only one tenth of the total execution time is contributed by the solver cycles on the GPU. Similarly, setup times of FSPAI are one eight of the solver cycle times for the CPU case, while they are almost of the same magnitude for the GPU case. Consequently, no benchmark including SPAI or FSPAI sparsity pattern updates is carries out, because this would only add to the setup time, which is already identified as the bottleneck in the GPU case.

5. CONCLUSIONS

We identified SPAI and FSPAI as attactive black-box preconditioners for the use on massively parallel computing architectures such as GPUs. Even though the number of solver iterations required to reach a certain relative residual are typically higher than for ILUT, overall execution times are reduced considerably for FSPAI due to the fully parallel application within the iterative solver. Since the preconditioner setup phase cannot benefit from GPU acceleration to the same extent as the solver iteration phase, it is increasingly important to keep preconditioner setup times low, possibly at the price of a higher number of iterative solver iterations. In particular, our benchmark results suggest that pattern updates for SPAI and FSPAI should better be avoided unless iterative solvers fail to converge.

We have addressed a high memory footprint for (F)SPAI by splitting the work load into chunks of sufficiently small size. Despite the high memory requirements of SPAI, the total solver execution time is still comparable to ILUT for our test case. It is expected that SPAI can consequently be a good choice for linear systems where ILUT does not perform well.

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition*. SIAM (2003).

[2] U. Trottenberg *et. al.*, *Multigrid*. Academic Press (2001).

[3] M. J. Grote and T. Huckle, Parallel Preconditioning with Sparse Approximate Inverses. *SIAM J. Sci. Comp.*, vol. 18, no. 3, p. 838–853 (1997).

[4] D. Göddeke *et al.*, Accelerating Double Precision FEM Simulations with GPUs. *Proc. ASIM* (2005).

[5] NVIDIA CUDA. `http://www.nvidia.com/`.

[6] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`.

[7] MAGMA library. Online: `http://icl.cs.utk.edu/magma/`

[8] ViennaCL. `http://viennacl.sourceforge.net/`.

[9] A. Kallischko *et al.*, Modified Sparse Approximate Inverses (MSPAI) for Parallel Preconditioning. PhD thesis. Technische Universität München (2008).

[10] N. Bell and M. Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004* (2008).

[11] M. M. Baskaran and R. Bordawekar, Optimizing Sparse Matrix-Vector Multiplication on GPUs. *IBM RC24704* (2008).

[12] OpenMP. `http://openmp.org/`.

[13] Boost C++ libraries. `http://www.boost.org/`.

[14] M. R. Hestenes and E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Stand.*, vol. 49, no. 6 (1952).

[15] T. Huckle, Factorized Sparse Approximate Inverses for Preconditioning. *J. Supercomput.*, vol. 25, p. 109–117 (2003).

[16] I. Kaporin, New Convergence Results and Preconditioning Strategies for the Conjugate Gradient Method. *Numer. Linear Algebra Appl.*, vol. 1, p. 179–210 (1994).

[17] H. A. van der Vorst, Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Non-Symmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, vol. 12, p. 631–644 (1992).