Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

JAKOB SCHNECK, MARTIN WEISER, FLORIAN WENDE

# Impact of mixed precision and storage layout on additive Schwarz smoothers

ZIB Report 18-62 (December 2018)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

# Impact of mixed precision and storage layout on additive Schwarz smoothers

Jakob Schneck, Martin Weiser, Florian Wende

January 8, 2019

**Abstract**

The growing discrepancy between CPU computing power and memory bandwidth drives more and more numerical algorithms into a bandwidth-bound regime. One example is the overlapping Schwarz smoother, a highly effective building block for iterative multigrid solution of elliptic equations with higher order finite elements. Two options of reducing the required memory bandwidth are sparsity exploiting storage layouts and representing matrix entries with reduced precision in floating point or fixed point format. We investigate the impact of several options on storage demand and contraction rate, both analytically in the context of subspace correction methods and numerically at an example of solid mechanics. Both perspectives agree on the favourite scheme: fixed point representation of Cholesky factors in nested dissection storage.

**Keywords:** higher order finite elements, mixed precision, overlapping Schwarz smoother
**MSC 2010: 65N55, 65Y10, 65Y20**

## 1 Introduction

The numerical solution of elliptic partial differential equations by finite element methods is an essential building block and one of the corner stones of scientific computing, with applications in diffusion processes, solid mechanics, and incompressible fluids [10, 26]. If more than moderate accuracy is requested, higher order finite elements are indispensable for an efficient solution process, with polynomial ansatz orders typically in the range from three to twelve depending on the problem's regularity properties. Despite higher order discretization, the arising linear systems tend to be rather large and sparse, such that, in particular in 3D problems, iterative solvers are preferable due to the superlinearly growing memory demand of direct sparse solvers.

The natural iterative solver for elliptic problems is the conjugate gradient method, preconditioned with multigrid or domain decomposition methods, leading to solvers of optimal complexity. Several variants of constructing multigrid hierarchies and smoothers exist [4, 23]. Here we focus on one particular scheme, where the hierarchy consists of a step from high-order finite elements to linear finite elements on the same grid, followed by a classical $h$-multigrid hierarchy. For this to work, the top level smoother must be highly effective. We consider a block-Jacobi smoother also known as overlapping Schwarz preconditioner, that

solves a small to moderately large and rather dense system for each grid node. The optimal complexity of this approach has been shown in [21] on quadrilaterals, and extended to simplicial meshes [22] and spectral elements [19].

While this overlapping block-Jacobi smoother is highly effective in terms of contraction rate, its application is significantly more expensive than a sparse matrix vector multiplication with the problem's stiffness matrix, and is therefore the most expensive part of the solver. During the last decades, the computing power of CPU cores has been growing faster than the memory bandwidth [18], and thus the smoother application is a bandwidth-bound operation. An acceleration of the smoother application therefore depends mainly on the reduction of the smoother storage size, and to some extent also on the access patterns.

Two of the main aspects affecting smoother storage size are sparsity of the Jacobi blocks and the representation of matrix entries. These are the aspects we investigate from both a theoretical and an experimental point of view here. Exploitation of sparsity is possible using standard sparse solver techniques [9], but, as the blocks are only moderately sparse, the trade-offs can be different from usual sparse solver situations.

Inexact representation of the Jacobi blocks offer additional or alternative means of reducing storage demands. One line of approach comprises structured and low-rank matrix approximations such as $\mathcal{H}$-matrices [16]. Another one is mixed precision arithmetics, leading to different representations of matrix entries as real numbers. This is attractive since the accuracy provided by the smoother need not be particularly high, and consequently, use of mixed precision arithmetics has been studied for HPC computing in general and the iterative solution of linear equation systems in particular in various contexts [2,3,7,12,15], in particular recently for non-overlapping block Jacobi solvers [1].

Beyond mixed precision representations, more sophisticated lossy compression schemes based on transform coding have been proposed for floating point values, see, e.g., [13, 17]. These do, however, incur significant computational work for decompression, which can outweigh the bandwidth savings. Therefore, we restrict the attention to simpler mixed precision representations here.

After defining the problem setting and the overlapping Schwarz smoother in Sec. 2, we discuss different smoother representations in Sec. 3, exploiting both sparsity and mixed precision and providing some theoretical insight for different ansatz order. Finally, numerical experiments and performance evaluations are reported in Sec. 4.

## 2 Overlapping Schwarz preconditioners

As a model problem, we consider the stationary elliptic PDE

$$\begin{aligned} -\operatorname{div}(\sigma \nabla u) &= f \quad \text{in } \Omega \\ n^T \sigma \nabla u + \alpha u &= \beta \quad \text{on } \partial\Omega \end{aligned} \tag{1}$$

on a polyhedral domain $\Omega \subset \mathbb{R}^d$. Let $\mathcal{T}$ be a simplicial triangulation of $\Omega$, with vertices $\mathcal{N}$ and mesh size $h = \min_{T\in\mathcal{T}} \operatorname{diam}(T)$. On $\mathcal{T}$ we define the ansatz space $V_p = \{u \in H^1(\Omega) \mid \forall T \in \mathcal{T} : u|_T \in \mathbb{P}_p\}$ of globally continuous and piecewise polynomial functions of order at most $p$. Choosing ansatz functions with smallest support as a basis to be used in a Galerkin discretization transforms (1) into a large, sparse, symmetric positive definite, and ill-conditioned

linear equation system

$$Ax = b. \tag{2}$$

Sparse direct solvers produce superlinear fill-in [8] in particular for 3D problems, and therefore induce a huge computational effort and often a prohibitive memory demand on large problems. If approximate solutions are sufficient, iterative solvers as the conjugate gradient method are a viable alternative, but suffer from the large condition number of $A$. For that reason, symmetric positive definite preconditioners $B \approx A$ are required, transforming (2) into the system

$$B^{-1}Ax = B^{-1}b \tag{3}$$

with smaller condition number $\kappa(B^{-1}A) \ll \kappa(A)$. In the optimal case, the condition number is independent of both $h$ and $p$, such that the number of CG-iterations required to reach a given accuracy is bounded independently of the problem size.

One such preconditioner is the overlapping additive Schwarz method with coarse space, analyzed first for quarilaterals in [21]. Let $\omega_\xi = \bigcup_{T \ni \xi} T$ denote the patch of simplices incident to the vertex $\xi$, $V_\xi = \{u \in V_p \mid \operatorname{supp} u \subset \omega_\xi\}$ the subspace of finite element functions with support contained in the patch $\omega_\xi$. Moreover, let the prolongation $P_\xi$ be a matrix representation of the embedding of $V_\xi$ into the ansatz space $V_p$. Local stiffness matrices $A_\xi = P_\xi^T A P_\xi$ are defined by Galerkin projection. Similarly, let $P_1$ represent the embedding $V_1 \subset V_p$ and $A_1$ be the corresponding stiffness matrix. Then the overlapping additive Schwarz preconditioner is

$$B^{-1} = P_1 A_1^{-1} P_1^T + \sum_{\xi \in \mathcal{N}} P_\xi A_\xi^{-1} P_\xi^T, \tag{4}$$

solving the problem simultaneously in all subspaces $V_\xi$, similar to a Jacobi method. Its optimality, i.e. the independence of the condition number $\kappa(B^{-1}A)$ of $h$ and $p$, has been shown in [22] in the framework of subspace correction methods [25], see also [10]:

**Theorem 2.1.** *Assume that the following conditions are satisfied.*

**Stability.** *There is a constant $K_1$ and for each $v \in V_p$ a decomposition $v = v_1 + \sum_\xi v_\xi$ with $v_1 \in V_1$ and $v_\xi \in V_\xi$ such that*

$$\langle v_1, A_1 v_1 \rangle + \sum_{\xi \in \mathcal{N}} \langle v_\xi, A_\xi v_\xi \rangle \leq K_1 \langle v, Av \rangle \tag{5}$$

*holds.*

**Strengthened Cauchy-Schwarz.** *There is $\gamma \in \mathbb{R}^{|\mathcal{N}| \times |\mathcal{N}|}$ such that*

$$\langle v_\xi, Av_\nu \rangle \leq \gamma_{\xi\nu} \langle v_\xi, A_\xi v_\xi \rangle^{\frac{1}{2}} \langle v_\nu, A_\nu v_\nu \rangle^{\frac{1}{2}} \tag{6}$$

*holds for all $v_\xi \in V_\xi$ and $v_\nu \in V_\nu$. Let $K_2 = 1 + \|\gamma\|$.*

*Then, the preconditioned system has bounded condition number $\kappa(B^{-1}A) \leq K_1 K_2$. The number of PCG iterations required to reach a relative accuracy of $\mathrm{TOL} < 1$ in the energy norm is bounded by $n_{\mathrm{it}} = -(\sqrt{K_1 K_2} + 1) \log(\mathrm{TOL}/2)$.*

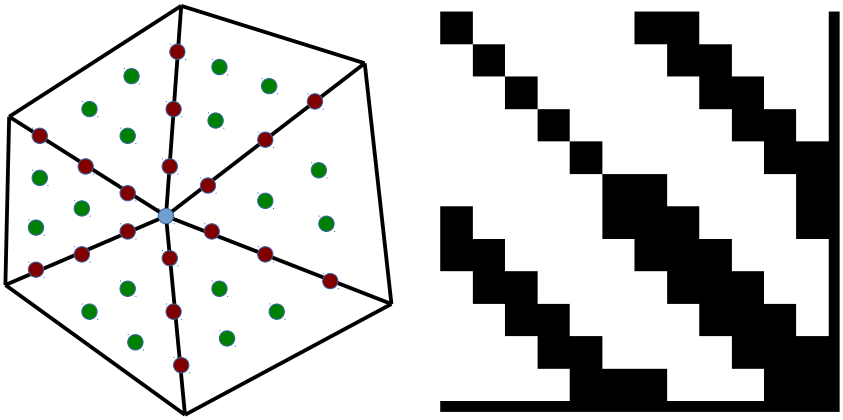

Figure 1: *Left:* 2D patch $\omega_\xi$ around a vertex. For $p = 4$, the 37 degrees of freedom associated to triangles, edges, and the center vertex are shown. *Right:* Sparsity pattern of the corresponding local stiffness matrix $A_\xi$ comprising mostly $3 \times 3$ blocks, in total 397 structurally nonzero entries (29% occupation).

The $P_1 A_1^{-1} P_1^T$ part of the preconditioner is responsible for independence of $h$, while $\sum_{\xi \in \mathcal{N}} P_\xi A_\xi^{-1} P_\xi^T$ leads to independence of $p$. In contrast, the effectivity of a nonoverlapping, purely diagonal or block-diagonal Jacobi preconditioner quickly degrades with growing $h$ or $p$.

**Remark 2.2.** Instead of an exact solve on the coarse space $V_1$ by $A_1^{-1}$, a suitable preconditioner $B_1^{-1}$ could be used. Examples are multigrid preconditioners like BPX [6] or multilevel domain decomposition methods [20]. Similarly, instead of the exact local solves with $A_\xi$, a finer decomposition of the ansatz space in more but smaller subspaces associated not only to patches around vertices, but also to patches around edges and faces can be used. This reduces the size of the local systems significantly, but tends to lead to larger though still bounded [22] or almost bounded [5] condition numbers.

The local matrices $A_\xi$ are moderately sparse, with a dimension growing as $\mathcal{O}(p^d)$. For 2D problems, on average six incident triangle to a vertex lead to an expected size of $3p^2$ (in leading order) and $\frac{3}{2}p^4$ nonzero entries in $A_\xi$, which leads to an occupation of the matrices of around 25%, see Fig. 1. In 3D problems with about 20 incident tetrahedra, the size is $\frac{10}{3}p^3$ with $0.56p^6$ nonzero entries, corresponding to a density of around 10%.

Already for moderate $p$, factorizing the $A_\xi$ is therefore the main computational effort during preconditioner construction, and the matrix-vector multipliation involving $A_\xi^{-1}$ the main effort during preconditioner application. In comparison, the effort for construction and application of $P_1 A_1^{-1} P_1^T$ is negligible. In the subsequent investigations on computational effort, we will therefore concentrate on the Schwarz preconditioner

$$B_p^{-1} = \sum_{\xi \in \mathcal{N}} P_\xi A_\xi^{-1} P_\xi^T \tag{7}$$

without coarse space.

# 3 Smoother storage organization and data representation

The application of the overlapping Schwarz preconditioner (7) consists of matrix-vector products $A_\xi^{-1} r_\xi$. On today's multicore CPUs, this is typically a memory bound operation if several threads are active, such that the computation power of floating point units and SIMD vector extensions such as SSE or AVX is largely unused.

The actual execution time, therefore, depends to a large extent on the amount of data and the storage organization used for representing $A_\xi^{-1}$.

Several different storage schemes are possible: (i) direct storage of all $A_\xi^{-1}$ using, e.g., LAPACK formats GE, SY, and P (ii) storage of $\sum_{\xi\in\mathcal{T}} P_\xi A_\xi^{-1} P_\xi^T$ as sparse matrix in, e.g., CRS format (iii) storing Cholesky factors $L_\xi$ of $L_\xi L_\xi^T = A_\xi$ in one of the LAPACK formats (iv) storing sparse Cholesky factors along with the pivoting table, using either CRS or a special purpose nested dissection format. Since the local stiffness matrices $A_\xi$ are of moderate sparsity and size, depending on $p$ and $d$, we will focus on factorizations and do not consider the iterative computation of $A_\xi^{-1} r_\xi$.

A second dimension of matrix storage that affects the amount of data is the real number representation. Widely used are the IEEE standard floating point representations of double (FP64) or single (FP32) precision, or the half precision (FP16) format that has become popular in machine learning. Besides those, special purpose floating point or fixed point formats with less accuracy and further reduced size may be considered. In contrast to the storage layout variants above, the arithmetic precision affects the accuracy of the preconditioner result, and may lead to an increased number of iterations or even divergence.

## 3.1 Storage layout

The different options of representing $A_\xi^{-1}$ sketched above lead to different size of data to be stored, but also to different memory access patterns. While the data size and its impact on run time and energy consumption can be estimated theoretically, the effect of different access patterns is hard to model. We will therefore only estimate the amount of data here and leave the impact of access patterns to Sec. 4 below.

As theoretical models of mesh connectivity within a patch $\omega_\xi$ we will use a hexagon in 2D and an icosahedron in 3D. Thus, we have matrix dimensions $n = 3p^2$ in 2D and $n = \frac{10}{3}p^3$ in 3D. All data sizes given in the text are in leading order of $p$ only, but are exact in Fig. 3.

**(i) Dense storage of $A_\xi^{-1}$.** The simplest approach is to compute the inverse, which is dense, and store it in one of the common LAPACK formats for symmetric matrices. The number of entries to be stored are $9p^4$ for GE layout, and $4.5p^4$ for the packed P format in 2D. In 3D, the respective numbers are $11.1p^6$ and $5.6p^6$. Symmetric storage SY is in between, as the memory demand is as for GE but only half of the entries are actually accessed, as in P. As may

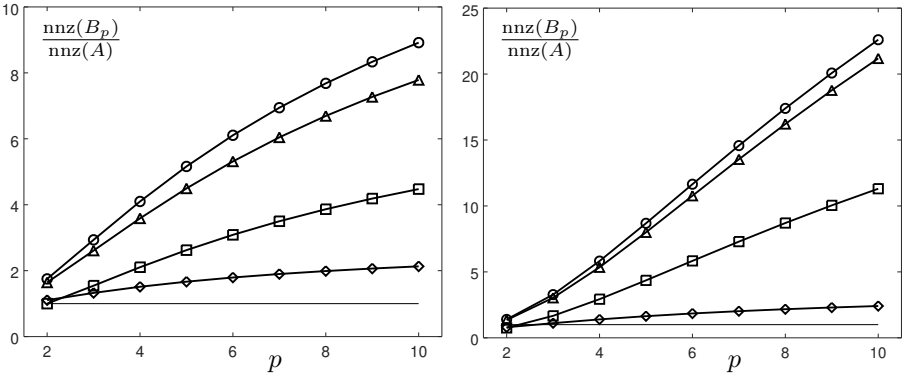


Figure 2: Memory requirement of different representations of $B_p$ relative to sparse storage of $A$ itself versus ansatz order $p$, in terms of number of matrix entries stored. ($\circ$) $\sum_\xi \operatorname{nnz}(A_\xi^{-1})$, ($\triangle$) $\operatorname{nnz}(B_p^{-1})$, ($\square$) $\sum_\xi \operatorname{nnz}(A_\xi^{-1,\mathrm{sym}})$, ($\Diamond$) $\sum_\xi \operatorname{nnz}(A_\xi)$. *Left:* 2D case *Right:* 3D case.

be expected, this naive approach is rather inefficient: the memory requirement of GE storage is 6–10 times larger than that for storing the stiffness matrix $A$ itself in 2D, and 15–35 times larger in 3D, see Fig. 2. Packed storage improves this by a factor of about two, still incurring a considerable overhead. Since in the CG method the number of applications of $B_p^{-1}$ and $A$ is the same, and the application is usually memory bound, reducing the storage size of $B_p^{-1}$ can be expected to speed up the solution process significantly.

**(ii) CRS storage of $B_p^{-1}$.** The local matrices $A_\xi$ overlap each other as submatrices of $A$, such that storing the inverses $A_\xi^{-1}$ individually wastes memory. Computing the sum $B_p^{-1} = \sum_{\xi\in\mathcal{T}} P_\xi A_\xi^{-1} P_\xi^T$ explicitly and storing it in a sparse matrix format fuses several entries and thus reduces the memory demand. Moreover, the application of $B_p^{-1}$ can be parallelized without locking on NUMA systems, as the overlap has already been resolved, and fewer accesses to the right hand side and solution vectors are performed.

The memory savings, however, are quantitatively small, see Fig. 2. The storage size in terms of matrix entries is reduced by about 10% in 2D and even less in 3D. Moreover, indices have to be stored for the entries, which adds some overhead.

**(iii) Dense storage of Cholesky factor $L_\xi$.** The dense storage of Cholesky factors in LAPACK formats GE, SY, and P involves the same amount of memory as the dense storage of $A_\xi^{-1}$. However, the entries need to be accessed twice in order to solve the system. If the factor is too large to fit into a cache, this incurs a larger amount of data to be read from memory. This makes dense storage of $L_\xi$ less attractive than dense packed storage of $A_\xi^{-1}$, except for that it is more robust with respect to inexact data representation, see Sec. 3.2.1.

**(iv) Sparse storage of Cholesky factor $L_\xi$.** In contrast to $A_\xi^{-1}$, the Cholesky factor $L_\xi$ is relatively sparse. The popular CRS format, however, incurs a certain overhead of indices to be stored alongside the entries of $L_\xi$, and leads to

irregular access patterns that may make the use of vector instructions less efficient. Since the local matrices $A_\xi$ and hence $L_\xi$ are only moderately sparse, special purpose block-structured approaches might be an attractive alternative, exploiting most of the potential sparsity while preserving SIMD-friendly access patterns.

One such method is nested dissection [11], with just one or two levels, decomposing the degrees of freedom into two disjoint sets of roughly the same size, separated by a small third set, which induces a large zero block in

$$L_\xi = \begin{bmatrix} L_1 & & \\ \mathbf{0} & L_2 & \\ S_1 & S_2 & L_S \end{bmatrix}.$$

The number of entries in $L_\xi$ for a single level of dissection is then in leading order $2.3p^4$ in 2D and $2.8p^6$ in 3D, reducing the memory demand by about a factor of two compared to packed storage of $A_\xi^{-1}$ and reducing the bandwidth demand to the same level.

Note that a suitable permutation into nested dissection format can be stored with two bits per degree of freedom and dissection level. If the blocks $S_1$ and $S_2$ are treated as dense, the inverse $L_\xi^{-1}$ has the same sparsity structure, such that $L_\xi^{-1}$ can be stored with the same amount of data.

A second option is static condensation [24], essentially a non-nested dissection approach with vertex, edge, and facet degrees of freedom forming the separator. The elimination of cell-associated degrees of freedom does not induce any fill-in. This leads to a block-sparse pattern of $L_\xi$ where fill-in is created only when degrees of freedom associated with edges or faces are eliminated. The resulting storage size is optimal in two leading orders of $p$: $0.75p^4$ in 2D and $0.28p^6$ in 3D. While this is asymptotically optimal for large $p$, the asymptotics sets in rather late, and the scheme creates many rather small dense blocks, for each of which the indices need to be stored. For small to moderate $p$, the access pattern is therefore less suited for SIMD processing than the nested dissection approach. Combining static condensation and one level of nested dissection can reduce the required storage further, in particular for small orders $p$, at the expense of creating even more and smaller blocks in $L_\xi$, as well as increasing implementation complexity.

Compared to the naive dense storage of $A_\xi^{-1}$, the sparse representations of $L_\xi$ promise a significant reduction of storage size as shown in Fig. 3, and hence shorter run times and energy consumption. Still, the amount of data to be read from memory is about 2–5 times larger than for the stiffness matrix.

## 3.2 Mixed precision storage on CPUs

The sparser storage schemes for $B_p$ reduce the amount of data simultaneously with the number of floating point operations, but still require 3–5 times the memory size compared to the global stiffness matrix $A$ for usual ansatz orders. A further reduction in memory size is possible with smaller representation of matrix entries, leading to mixed precision approaches. As in [1], we focus on the *storage* of the preconditioner in reduced precision, assuming that the factorization of $A_\xi$ is performed in exact arithmetics, or at least sufficiently accurate such that the errors are negligible. Unfortunately, reduced precision storage affects the quality and hence the effectivity of the preconditioner.

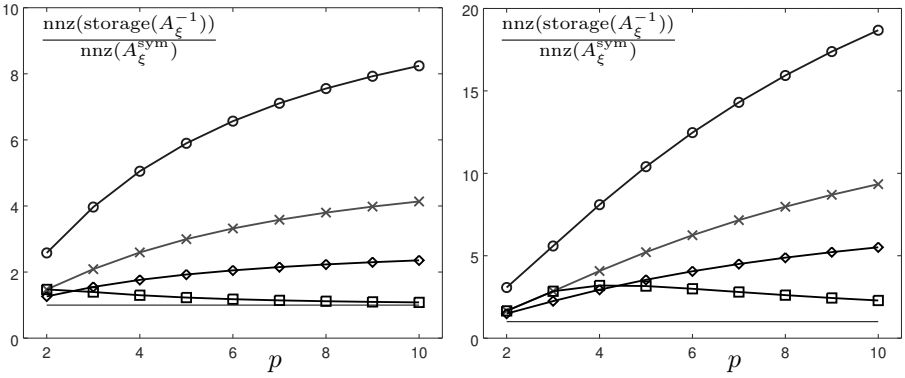


Figure 3: Matrix entries stored for different representations of $A_\xi^{-1}$ relative to $\mathrm{nnz}(A_\xi^{\mathrm{sym}})$, i.e. symmetric sparse storage of $A_\xi$, versus ansatz order $p$, in terms of number of matrix entries stored. ($\circ$) $\mathrm{nnz}(A_\xi^{-1})$, ($\times$) $\mathrm{nnz}(A_\xi^{-1,\mathrm{sym}})$, ($\Diamond$) one level of nested dissection, ($\square$) static condensation. *Left:* 2D case *Right:* 3D case.

### 3.2.1 Impact of reduced precision storage.

In the following we will derive bounds on the representation error such that the efficiency of the preconditioner, in terms of a bound on the iteration count, deteriorates only by a certain factor. Unless stated otherwise, matrix norms will be the spectral norm.

**Lemma 3.1.** *In addition to the assumptions of Thm. 2.1, let* $a, b \geq 1$ *such that*

$$a^{-1}\langle v, A_\xi v\rangle \leq \langle v, \tilde{A}_\xi v\rangle \leq b\langle v, A_\xi v\rangle \tag{8}$$

*holds for all* $\xi$ *and* $v \in V_\xi$. *Then there are* $\alpha, \beta \in [0,1]$, *such that the preconditioner* $\tilde{B}^{-1} = P_1 A_1^{-1} P_1^T + \sum_{\xi\in\mathcal{N}} P_\xi \tilde{A}_\xi^{-1} P_\xi^T$ *leads to a condition number bounded by*

$$\kappa(\tilde{B}^{-1}A) \leq s^2 K_1 K_2, \quad s^2 = (1-\alpha+\alpha a)(1-\beta+\beta b). \tag{9}$$

*The bound on the iteration count is*

$$\tilde{n}_{\mathrm{it}} = \frac{s\sqrt{K_1K_2}+1}{\sqrt{K_1K_2}+1} n_{\mathrm{it}} \leq s\, n_{\mathrm{it}}. \tag{10}$$

*Proof.* In order to apply Thm. 2.1 to $\tilde{B}$ instead of $B$, we bound

$$\langle v_1, A_1 v_1\rangle + \sum_{\xi\in\mathcal{N}} \langle v_\xi, \tilde{A}_\xi v_\xi\rangle \leq \langle v_1, A_1 v_1\rangle + b\sum_{\xi\in\mathcal{N}} \langle v_\xi, A_\xi v_\xi\rangle \leq (1-\beta+\beta b) K_1 \langle v, Av\rangle$$

and

$$\langle v_\xi, Av_\nu\rangle \leq \gamma_{\xi\nu}\langle v_\xi, A_\xi v_\xi\rangle^{\frac{1}{2}} \langle v_\nu, A_\nu v_\nu\rangle^{\frac{1}{2}} \leq a\gamma_{\xi\nu}\langle v_\xi, A_\xi v_\xi\rangle^{\frac{1}{2}} \langle v_\nu, A_\nu v_\nu\rangle^{\frac{1}{2}}.$$

Consequently, we obtain

$$\begin{aligned}\kappa(\tilde{B}^{-1}A) &\leq (1-\beta+\beta b)K_1(1+a\|\gamma\|) \\ &= (1-\beta+\beta b)K_1(1-\alpha+\alpha a)K_2 = s^2 K_1 K_2\end{aligned}$$

and thus (9). From that, (10) follows directly from Thm. 2.1. $\square$

In Lem. 3.1 above, $\alpha = \beta = 1$ is always an admissible choice. Depending on the problem, its discretization, and the space splitting, smaller admissible values may exist. These will, however, be hard to obtain a priori.

**Remark 3.2.** The factor $s$ of Lemma 3.1, which will also show up in Thms. 3.3 and 3.4 below, is to be interpreted with care. It does not describe a bound or estimate for the increase in iteration count due to inexact storage, but a bound for the increase of the *theoretical upper bound of the iteration count*. While this indeed provides a bound for the total iteration count, it does not bound the ratio of the actual iteration counts.

**Error bounds for storing $A_\xi^{-1}$.** Instead of storing $A_\xi^{-1}$, we store $\tilde{A}_\xi^{-1} = A_\xi^{-1} + \delta_\xi$ with symmetric rounding error matrix $\delta_\xi$.

**Theorem 3.3.** *Let the representation errors $\delta_\xi = \tilde{A}_\xi^{-1} - A_\xi^{-1}$ be bounded by*

$$\|\delta_\xi\| \leq \epsilon \|A_\xi\|^{-1} \tag{11}$$

*for some $\epsilon < 1$. Then the preconditioner $\tilde{B}$ is positive definite and there are $\alpha, \beta \in [0,1]$ such that its bound of the PCG iteration count satisfies $\tilde{n}_{\text{it}} \leq s n_{\text{it}}$ with*

$$s^2 = (1+\alpha\epsilon)\left(1 + \frac{\beta\epsilon}{1-\epsilon}\right). \tag{12}$$

*Proof.* For brevity, we drop the subscript $\xi$. Due to $A$ being positive definite and $\delta$ being symmetric, $A\delta$ has a real spectrum. Assumption (11) implies

$$\lambda_{\max}(I + A\delta) \leq 1 + \|A\|\|\delta\| \leq 1 + \epsilon =: a$$

and similarly $\lambda_{\min}(I+A\delta) \geq 1-\epsilon =: b^{-1}$. With $\tilde{A} = (A^{-1}+\delta)^{-1}$ and $w = A^{1/2}v$, (8) is equivalent to

$$\begin{aligned} a^{-1}\|w\|^2 \leq (\lambda_{\max}(I + A\delta))^{-1}\|w\|^2 \\ \leq \langle w, (I + A^{1/2}\delta A^{1/2})^{-1} w\rangle \\ \leq (\lambda_{\min}(I + A\delta))^{-1}\|w\|^2 \leq b\|w\|^2 \end{aligned}$$

for all $w$. Thus, (8) is satisfied for all $\xi$. Applying Lemma 3.1 yields

$$\begin{aligned} \tilde{n}_{\text{it}}^2 &\leq (1 - \alpha + \alpha a)(1 - \beta + \beta b)\, n_{\text{it}}^2 \\ &= (1+\alpha\epsilon)\left(1 + \frac{\beta\epsilon}{1-\epsilon}\right) n_{\text{it}}^2 \end{aligned}$$

and hence the claim. □

For a small predicted increase in iteration count, i.e. $s \approx 1$, assumption (11) is roughly the *relative* accuracy requirement

$$\frac{\|\delta_\xi\|}{\|A_\xi^{-1}\|} \leq 2\frac{s-1}{\kappa(A_\xi)(\alpha+\beta)} \tag{13}$$

in terms of the condition number of $A_\xi$, similar to the findings in [1].

We observe, however, that the accuracy is relative to $\|A_\xi^{-1}\|$, and not component-wise. This suggests that a fixed point number representation is sufficient for storage of $A_\xi^{-1}$, and even more memory-efficient since no individual exponents need to be stored.

**Error bounds for storing $L_\xi$.** Now we store $\tilde{L}_\xi = L_\xi + \delta_\xi$ instead of $L_\xi$, again with rounding matrix $\delta_\xi$.

**Theorem 3.4.** *Let the representation errors $\delta_\xi = \tilde{L}_\xi - L_\xi$ be bounded by*

$$\|\delta_\xi\| \leq \epsilon \|L_\xi^{-1}\|^{-1} \tag{14}$$

*for some $\epsilon < 1$. Then the preconditioner $\tilde{B}$ is positive definite and there are $\alpha, \beta \in [0,1]$ such that its bound of the PCG iteration count satisfies $\tilde{n}_{\text{it}} \leq s n_{\text{it}}$ with*

$$s^2 = \left(1 + \alpha\epsilon \frac{2-\epsilon}{(1-\epsilon)^2}\right)(1 + \beta\epsilon(2+\epsilon)) \tag{15}$$

*Proof.* Again we omit the subscript $\xi$. Let $\Lambda = (I + L^{-1}\delta)(I + L^{-1}\delta)^T$, $a^{-1} := (1-\epsilon)^2 = (1 - \|L^{-1}\|\|\delta\|)^2 \leq \lambda_{\min}(\Lambda)$, and $b := (1+\epsilon)^2 = (1 + \|L^{-1}\|\|\delta\|)^2 \geq \lambda_{\max}(\Lambda)$. Then,

$$a^{-1}\|w\|^2 \leq \langle w, L^{-1}(L+\delta)(L+\delta)^T L^{-T} w\rangle \leq b\|w\|^2.$$

holds for all $w$ and implies (8) with $v = L^{-T}w$. Due to (14),

$$\begin{aligned} s^2 &= (1 - \alpha + \alpha a)(1 - \beta + \beta b) \\ &= \left(1 - \alpha + \frac{\alpha}{(1-\epsilon)^2}\right)(1 - \beta + \beta(1+\epsilon)^2) \\ &= \left(1 + \alpha\epsilon\frac{2-\epsilon}{(1-\epsilon)^2}\right)(1 + \beta\epsilon(2+\epsilon)) \end{aligned}$$

holds, and applying Lemma 3.1 yields $\tilde{n}_{\text{it}} \leq s n_{\text{it}}$. □

As before, for $s \approx 1$ the required normwise relative precision is governed by the condition number as

$$\frac{\|\delta_\xi\|}{\|L_\xi\|} \leq 2\frac{s-1}{\kappa(L_\xi)(\alpha+\beta)}. \tag{16}$$

Due to $\kappa(A_\xi) = \kappa(L_\xi)^2 \geq 1$, the accuracy requirement (16) is less restrictive than (13). Since $\kappa(A) \geq \mathcal{O}(p^2)$, using Cholesky factors should gain efficiency over using inverses in terms of reduced precision storage for growing ansatz order $p$.

### 3.2.2 Floating point representation

One widespread technical standard for floating point arithmetic has been specified by the *Institute of Electrical and Electronics Engineers* (IEEE) in 1985, named *IEEE 754*. It comprises rules for floating point arithmetic, rounding and exchange of floating point numbers as well as the definition of their bit representation. The implementation of this standard is platform dependent, and current computer architecturs have support for the full original standard and its extension IEEE 754-2008 to a certain (and varying) extent only. Emerging architectures like General Purpose Graphics Processing Units (GPGPUs) and ARM push forward the support for some of the latest specifications. Along with this development and the adaptation of widely used programming languages, reduced precision floating point and fixed point arithmetic becomes increasingly

| | | load/store | processing | supporting platform/architecture* |
|---|---|---|---|---|
| $\mathrm{FP}_{11,52}$ | (double type) | × | × | x86, ARM, GPU |
| $\mathrm{FP}_{8,23}$ | (single type) | × | × | x86, ARM, GPU |
| $\mathrm{FP}_{5,10}$ | (half type)** | × | | x86 |
| | | × | × | ARM, GPU |
| $\mathrm{FP}_{R<11,P<52}$ (general) | | × | | x86, ARM, GPU |

Table 1: Overview of the supported floating point formats on x86 (Intel), ARM and (GP)GPU platforms/architectures. In case of x86 and (GP)GPU, we consider the latest server/workstation product line: Intel Skylake, Nvidia Tesla P100, AMD VEGA.

* Support might vary with the actual model.

** There is no half type in the C/C++ programming language. Extensions like OpenCL or CUDA, however, provide a half type on supporting platforms. On x86 platforms with FP16C support, conversions between the single/double and half type can happen through intrinsic functions.

relevant, and we are dealing with it not just on GPGPUs and ARM processors, but also on standard x86 CPU based systems in consequence.

The IEEE 754 floating point standard specifies the number of bits in the exponent ($R$) and the mantissa ($P$) to be $R_s = 8$ and $P_s = 23$ for the *single* type (`float`), and $R_d = 11$ and $P_d = 52$ for the *double* type (`double`), respectively. Beside the cases where the accuracy provided by either the single or the double type is insufficient, there is those where not all bits are needed to hold a given value, e.g., if a loss of accuracy is acceptable within some limits. Based on the single/double type, any representation $\mathrm{FP}_{R,P}$ with $R \leq R_{s/d}$ and/or $P \leq P_{s/d}$ can be easily implemented by a sequence of arithmetic and bit operations together with an optional rescaling so as to cope with the range of $\mathrm{FP}_{R,P}$. On most general purpose computer platforms, however, these floating point representations cannot be processed in hardware. Implementing the conversion between the single/double type and $\mathrm{FP}_{R,P}$ adds to the actual calculation and has to be integrated on a fine-grained level because otherwise data is moved twice. The benefit thus lies only in the reduced memory foot print as a result of the implicit data compression.

Table 1 gives a brief overview of the supported floating point formats on x86 (Intel), ARM and (GP)GPU platforms/architectures. While the latter two architecturally allow for *half* type processing (depending on the actual model), support for the half type on x86 platforms is restricted to conversion combined with load/store, and can be used through intrinsic functions only.

For general $\mathrm{FP}_{R\leq 11,P\leq 52}$ formats, bits can be packed into suitably large fundamental C/C++ data types or bitstreams to implement the load/store operation. Listing 1 shows a possible implementation of the compress (and store) operation that processes a sequence of $n$ floating point numbers of type `T`, packs their $\mathrm{FP}_{R,P}$ representations into 64-bit words in groups of size $64/(1 + R + P)$, and writes these packs to the output until all words are done. The `down_convert<T, R, P>` function implements the elementwise conversion (or recoding) from `T` to $\mathrm{FP}_{R,P}$. The (load and) decompress operation works the

```
1  template <typename T, size_t R, size_t P>
2  void compress(const T* in, void* out, const size_t n)
3    static_assert(is_floating_point<T>::value, "T NOT SUPPORTED");
4    // total number of bits (input)
5    constexpr size_t bits_in = sizeof(T) * 8;
6    // total number of bits (output): sign+exponent+mantissa
7    constexpr size_t bits_out = 1 + R + P;
8    static_assert(bits_out < bits_in, "NO COMPRESSION POSSIBLE");
9    // data type for the packing
10   using pack_t = int64_t;
11   pack_t* pack_out = reinterpret_cast<pack_t*>(out);
12   constexpr size_t pack_n = (sizeof(pack_t) * 8) / bits_out;
13   // process the input with chunk size 'pack_n'
14   for (size_t i = 0, j = 0; i < n; i += pack_n, ++j){
15     pack_t pack = 0;
16     for (size_t ii = 0; ii < min(pack_n, n-i); ++ii){
17       pack_t pack_in = down_convert<T, R, P>(in[i+ii]);
18       pack |= (pack_in << (ii * bits_out));
19     }
20     pack_out[j] = pack;
21   }
22 }
```

Listing 1: Pseudo code for the compression of an input data stream of $n$ floating point numbers of type `T` into an output data stream using 64-bit word packs, each holding multiple compressed words.

other way around using the `up_convert<T, R, P>` function to convert elementwise from $\mathrm{FP}_{R,P}$ to `T`.

A reduced exponent range due to $R \ll 8$ can lead to overflow and hence huge representation errors. Despite large exponents, the entries may nevertheless be of comparable magnitude. Local mesh widths or PDE coefficients, for instance, tend to affect all the entries in a similar way. Thus, a rescaling of the input $F = \{f_1, \ldots, f_n \,|\, f_i \in \mathbb{R}\}$ can be applied before and after the elementwise conversion in the compress and decompress function, respectively. For that to work, the input has to be scanned first in the compress function to find the maximum absolute value $f_{\max} = \max_{f \in F}(|f|)$. The rescaling in the compress function then happens with $s = l_{R,P}/f_{\max}$ (not implemented in Listing 1), where $l_{R,P}$ is the largest number that can be represented with $\mathrm{FP}_{R,P}$.[1] In the decompress function the rescaling factor $1/s$ is used.

From a performance perspective, Listing 1 can serve as the default case for general $R$ and $P$ values. However, it will most likely suffer from the recoding of the exponent and mantissa (in `[up|down]_convert`) using bit operations, and specializations for specific $R$ and $P$ values that enable for simplifications and/or advanced optimizations are needed to gain performance over using `T`. Listing 2 implements the conversion from a C/C++ single type (`float`) to $\mathrm{FP}_{8,7}$ (or `bfloat16` in the field of *AI* and *Deep Learning*) by removing the lower 16 bits of the mantissa. The elementwise down-conversion and the packing of the $\mathrm{FP}_{8,7}$ representations into 64-bit words, is replaced by accessing the input sequence

[1] Adapting the IEEE 754 convention to $\mathrm{FP}_{R,P}$, we propose $l_{R,P} = (1 - 2^{-P-1})2^{2^{R-1}}$.

```
template <>
void compress<float, 8, 7>(const float* in, void* out, const size_t n)
  const int16_t* ptr_in = reinterpret_cast<const int16_t*>(in);
  int16_t* ptr_out = reinterpret_cast<int16_t*>(out);
  for (size_t i = 0; i < n; ++i)
    ptr_out[i] = ptr_in[2 * i + 1];
}
```

Listing 2: Template specialization of the `compress` function in Listing 1 for $R = 8$ and $P = 7$. The compression basically renders into accessing the input sequence with stride-2 and writing to the output with stride-1.

with stride-2 and writing to the output with stride-1. In the corresponding decompress function, the strided data access is the opposite, and the lower 16 bits of each output element are zero-filled.

#### 3.2.3 Fixed point representation

According to the analysis in Sec. 3.2.1, fixed point representations might be a preferable alternative to floating point types. For fixed point representations with $k > 0$ bits, we define the quantization function

$$q : [c_{\min}, c_{\max}] \to [0, 2^k - 1] \cap \mathbb{N}, \quad c \mapsto \begin{cases} \left\lfloor 2^k \frac{c - c_{\min}}{c_{\max} - c_{\min}} \right\rfloor, & c < 1 \\ 2^k - 1, & c = c_{\max} \end{cases} \tag{17a}$$

and its dequantization

$$q^+ : [0, 2^k - 1] \cap \mathbb{N} \to [c_{\min}, c_{\max}], \quad l \mapsto \frac{l + \frac{1}{2}}{2^k}(c_{\max} - c_{\min}) + c_{\min}. \tag{17b}$$

The quantized values can be stored as integers with $k$ bits, and the quantization error is bounded by $|q^+(q(c)) - c| \leq 2^{-(k+1)}(c_{\max} - c_{\min})$. Using the fixed point approximation $\tilde{C} := q^+(q(C))$ with $c_{\min} = \min_{ij}(c_{ij})$, $c_{\max} = \max_{ij}(c_{ij})$ for any dense matrix $C \in \mathbb{R}^{m \times m}$, we obtain

$$\|\tilde{C} - C\| \leq \|\tilde{C} - C\|_F \leq m 2^{-(k+1)}(c_{\max} - c_{\min}) \leq 2^{-k} m \|C\|. \tag{18}$$

Note that the factor one between spectral norm and Frobenius norm in the first inequality is sharp, but on average, a much smaller ratio of $\mathcal{O}(m^{-\frac{1}{2}})$ is expected, leading to the better average case estimate

$$\|\tilde{C} - C\| \leq 2^{-k} \sqrt{m} \|C\|. \tag{19}$$

## 4 Numerical experiments

In the following, we will investigate how the expected speedup translates into practice. First we analyze the run times of BLAS level 2 operations using reduced precision representations, and then turn to run times and iteration counts of additive Schwarz preconditioners.

### 4.1 BLAS level 2 operations with reduced precision

Among the performance critical matrix-vector operations used for the implementation of the preconditioner is *general matrix-vector multiplication* (*gemv*), *triangular (packed) matrix-vector multiplication* (*tpmv*), *symmetric (packed) matrix-vector multiplication* (*spmv*), and *triangular (packed) solve* (*tpsv*). Library implementations of these BLAS level 2 operations, e.g., Intel MKL, provide interfaces that allow for single and double data type processing. Using these implementations together with the reduced precision floating or fixed point representations therefore requires an upconversion (decompression) step before the actual application of the matrix-vector operation.

In order not to spill the cache during decompression, a block representation of the compressed matrix $\tilde{C}$ is required. That is,

$$\tilde{C} = (\tilde{C}_{ij}^{\square}) = ((\tilde{c}_{kl}^{\square})_{ij}) \tag{20}$$

with $\tilde{C}_{ij}^{\square} \in \mathbb{R}^{m^{\square} \times m^{\square}}$ for $i, j = 1, \ldots, \lceil m/m^{\square} \rceil$. BLAS operations on $\tilde{C}$ then become a sequence of BLAS operations on these blocks. The block size $m^{\square}$ should be of moderate size so that the blocks fit into the CPU cache. Otherwise, data would be moved from main memory into main memory again while decompressing, thereby rendering the benefit of the compression zero. The block representation has the additional benefit that the range $c_{\max} - c_{\min}$ of the entry values can be stored individually for each block, resulting in higher accuracy in (18) compared to a global quantization, at the expense of storing two additional floating point numbers per block. However, blocks should not be too small, as the overhead for calling into the BLAS library increases with the number of blocks, and processing too small blocks might drop the performance. The block size $m^{\square}$ hence is target for machine and context specific optimization.

For BLAS level 2 operations it is known that they are bandwidth-bound, so that implementing them ourselves should not hurt the performance that much compared to optimized library implementations. For fixed point representations with $k$ bits, we can directly operate on the integers if $k$ matches their word size. The matrix-vector multiplication $a \leftarrow \tilde{C}b$ (with the blocking induced by (20)) then reads

$$\begin{aligned} a_i^{\square} &= \sum_{j=1}^{\lceil m/m^{\square} \rceil} \tilde{C}_{ij}^{\square} b_j^{\square} \\ &= \sum_{j=1}^{\lceil m/m^{\square} \rceil} q^{+}(q(C_{ij}^{\square})) b_j^{\square} \\ &= \sum_{j=1}^{\lceil m/m^{\square} \rceil} \left( r_{ij} q(C_{ij}^{\square})) b_j^{\square} + p_{ij} \mathbb{1}^T b_j^{\square} \right) \end{aligned} \tag{21}$$

with

$$r_{ij} = 2^{-k}\big((c_{\max})_{ij} - (c_{\min})_{ij}\big), \quad p_{ij} = \left(\frac{r_{ij}}{2} + (c_{\min})_{ij}\right)$$

according to (17b), and $a_i^{\square}$ and $b_j^{\square}$ being the segments of $a$ and $b$ needed for the blockwise BLAS level 2 operations.

Fig. 4 shows the performance of the aforementioned BLAS level 2 operations on an Intel Xeon Gold 6138 (Skylake) compute node ($2 \times 40$ logical CPU cores, 192 GiB main memory, and 80 threads running) for matrices of size $m \times m$ with $m = 447$ (a *small* matrix that is not a power of 2) and $m = 2048$ (a *large* matrix), varying block size $m^{\square}$, and different floating point and fixed point representations. Floating point representations use Intel's MKL 2019 for double type BLAS level 2 operations on the blocks (after an explicit upconversion from either $\mathrm{FP}_{8,7}$ or $\mathrm{FP}_{8,23}$ to $\mathrm{FP}_{11,52}$), whereas an implementation of (21) is used for the fixed point representations.

The GFLOP values stated do not include any additional floating point operations that originate from the decompression, thereby allowing for a direct and fair (black box-like) performance comparison between the different representations. The gains over reference executions on the entire matrices using Intel's MKL 2019 are given as well. $\mathrm{FP}_{R,P}$ representations that use the packing scheme presented in Listing 1 give at most a factor 1.6 performance gain over the MKL reference, independently of the values of $R$ and $P$. They are not included in Fig. 4 for that reason. For the other representations, there are some interesting observations:

First, the block representation of the matrix using the double type results in a small performance drop over the reference execution if block sizes are either too small or too large. Choosing $16 \leq m^{\square} \leq 128$ on our Skylake compute node, the additional calling overhead and the decompression, which for the double type is a simple mem-copy, can be overcompensated by an increased data locality. However, in most cases, we observe no significant performance increase over the reference.

Second, the reduced precision floating point representations suffer to some extent from the explicit decompression of the blocks to $\mathrm{FP}_{11,52}$ before applying the actual BLAS operation. While both the compressed block and the decompressed block have to fit into the CPU cache in this case, for the fixed point representations only the compressed block is cached as the decompression happens on the fly with our implementation. The latter results in a better cache utilization that allows for larger blocks without causing performance drops due to spilling. In some cases, we see performance gains larger than the expected ones with the 16-bit fixed point representation for sufficiently large matrices.

Third, the 8-bit fixed point representation does not gain the performance significantly beyond the 16-bit case, which for matrices of size $2048 \times 2048$ is 200 – 380 GFLOPS on our Skylake compute node. The actual number of FLOPS that are executed is higher, as this value does not include the rescaling, which adds one fused-multiply-add (FMA) operation per matrix element. Taking that into account, a more realistic estimate of the GFLOPS should be at least a factor two higher than the values stated, which is between 15% and 30% peak performance.[2] The decreased memory footprint together with the blocking, thus render the BLAS level 2 operations significantly into the compute-bound regime. As the actual BLAS operation happens with $\mathrm{FP}_{11,52}$ floating point numbers, a performance gain over the 16-bit fixed point representation can result only

[2] We compiled our benchmark codes with the GNU 8.2 C++ compiler with optimizations for the Skylake architecture enabled. The AVX-512 base frequency of the Xeon Gold 6138 is 1.9 GHz in turbo mode with all cores used. With its 2 AVX-512 FMA units per CPU core, our compute node should give 2 (CPUs) · 20 (CPU cores) · 2 (FMA units) · 2 (FMA) · 8 (AVX-512, double type) · 1.9 GHz $\approx$ 2430 GFLOPS peak performance.

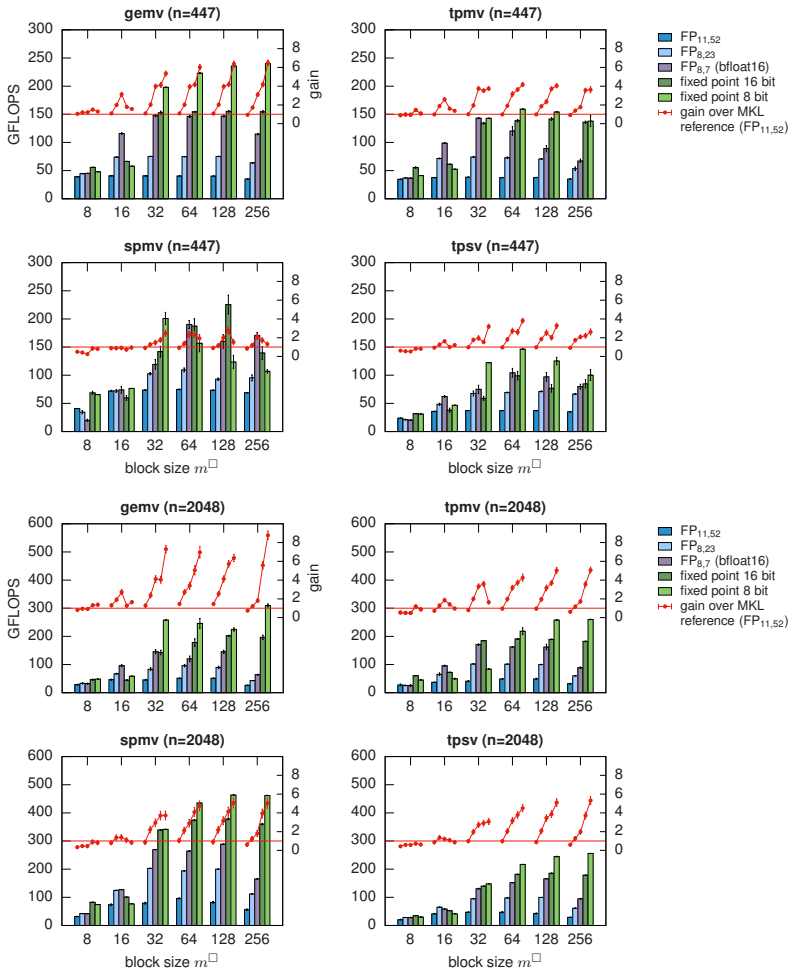


Figure 4: Performance of BLAS level 2 operations *gemv*, *tpmv*, *spmv* and *tpsv* (see text) on an Intel Xeon Gold 6138 (Skylake) compute node ($2 \times 40$ logical CPU cores and 80 threads running) for matrices of size $m \times m$ with $m = 447$ and $m = 2048$, different floating and fixed point representations and varying block size $m^{\square}$.

from the further reduced memory footprint and/or an improved upconversion. The latter, however, involves almost the same number of data-parallel (SIMD) operations in both the 8-bit and 16-bit case. Significant gains over the reference thus are not to be expected with the 8-bit instead of the 16-bit fixed point represenation.

Summarizing, the block size $m^{\square}$ has a major impact on the achievable performance of the BLAS level 2 operations. For the reduced floating point representations, too small blocks mean more calling overhead into and reduced performance of the BLAS calls, whereas too many large blocks might not fit into the CPU cache. Our implementations operating directly on the fixed point representations avoid the explicit decompression step before the BLAS call, but combine it with the actual computation. A minimum block size of $m^{\square} = 32$

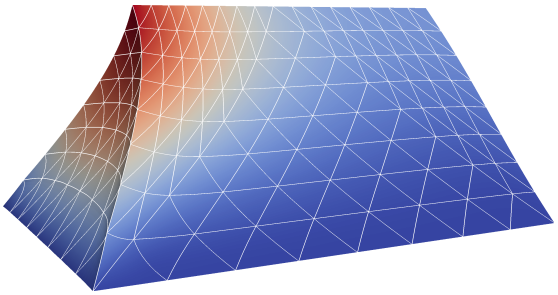

Figure 5: Cut-out of one quarter of the deformed domain of test problem (22). Color coding by displacement magnitude.

seems to be necessary on our Skylake compute node to get roughly 4x gain with 16-bit words instead of the 64-bit reference in that case.

## 4.2 Additive Schwarz preconditioners with reduced precision storage

**Test problem setup.** As a simple yet sufficiently interesting example for the application of an additive Schwarz smoother we consider the Lamé-Navier equation of linear solid mechanics,

$$-2\mu\Delta u - \lambda\nabla \operatorname{div} u = f \quad \text{in } \Omega = ]0,1[^3. \tag{22}$$

As material parameters we use those of steel, i.e. $\mu = 79.3{\cdot}10^9$ and $\lambda = 107.1{\cdot}10^9$. Boundary conditions are natural conditions on the right face of the unit cube, constant normal force applied to the left face, and Dirichlet conditions on the remaining sides, the latter ones implemented as penalty. The coarse grid is a symmetric splitting of the unit cube into 24 tetrahedra, and is uniformly refined three times, resulting in 2465 vertices. The resulting deformation is shown in Fig. 5.

As storage schemes, five versions have been implemented for numerical experiments. As a reference we use dense storage of $A_\xi^{-1}$ in double precision GE format, using standard BLAS *gemv*. In detail we consider block formats with various block sizes $m^\square$ and different representations as discussed in Sec. 4.1 for (a) dense non-symmetric storage of $A_\xi^{-1}$, (b) symmetric packed storage of $A_\xi^{-1}$, (c) packed storage of $L_\xi$, and (d) single level (and hence strictly speaking non-nested) dissection with packed storage of the resulting blocks in $L_\xi$.

As mentioned in Rem. 2.2, a BPX preconditioner would have been a natural choice for the coarse problem of P1 finite elements in (4). The overall contraction rate, however, is then dominated by the BPX part, such that inaccuracies in the overlapping Schwarz smoother are hidden. We therefore use a direct solver for the coarse problem, which yields an overall contraction rate around 0.31. The PCG method used for solving (22) is started with a zero initial iterate, and terminated based on the estimated energy error, cf. [10]. Test setup and preconditioner have been implemented using the Kaskade 7 toolbox [14].

**Iteration count.** First we check the theoretical impact of inaccurate storage on PCG iterations. The increase of upper bounds on iteration counts given

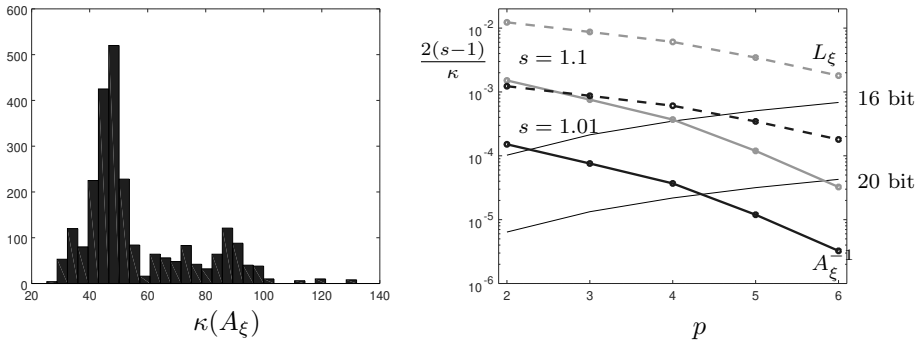


Figure 6: *Left:* Histogram of local matrix condition numbers $\kappa(A_\xi)$ for order $p = 3$. *Right:* Sufficient relative accuracy $\|\delta_\xi\|/\|A_\xi^{-1}\| \leq 2(s-1)/\kappa$ versus ansatz orders $p$ for different iteration bound factors $s$. Dashed lines for storing $L_\xi$, solid lines for storing $A_\xi^{-1}$. Increasing lines denote the relative accuracy expected with 16 and 20 bit fixed point representation.

in Thms. 3.3 and 3.4 depends on the relative accuracy of storage versus the condition number of the local matrices. The distribution of the latter is shown in Fig. 6, left. The apparent bimodality of the distribution is due to the penalty treatment of Dirichlet boundary conditions, and becomes more distinct for larger penalty factors.

Based on the maximum condition number observed for different ansatz orders $p$, we derive relative accuracies $2(s-1)/\kappa$ according to (13) and (16), respectively, for the conservative choice $\alpha = \beta = 1$. Relative accuracies sufficient to guarantee an increase of the iteration count bound by 1% and 10% corresponding to $s = 1.01$ and $s = 1.1$, respectively, are shown in Fig. 6, right, as decreasing curves. The relative accuracy expected from storage of entries in $k$-bit fixed point representation according to (19) is plotted for $k = 16$ and $k = 20$. These curves increase since the local matrix size grows as $p^3$. The figure shows that up to $p = 6$, a negligible iteration count increase of less than 1% can be expected when storing Cholesky factors $L_\xi$ with 20 bits fixed point accuracy, whereas 16 bit accuracy can be expected to lead to at most 10% increase in iteration count. Obviously, due to slower growth of condition number, storing $L_\xi$ is preferable compared to storing $A_\xi^{-1}$.

In order to investigate the impact of inaccuracy on actual iteration numbers, we store $A_\xi^{-1} + \gamma\delta_\xi$ in double precision, where the entries of $\delta_\xi$ are uniformly and independently distributed in $[-1, 1]$, and the scaling $\gamma = \epsilon\|A_\xi\|\,\|\delta_\xi\|$ ensures that the assumptions of Thm. 3.3 are satisfied. The distribution of entries of $\delta_\xi$ mimics the quantization error of fixed point representation. For different values of $\epsilon \in [0, 1[$ and orders $p \in \{3, 4\}$, the iteration count $n_{\mathrm{it}}(s)$ for reaching a relative accuracy of $\mathrm{TOL} = 10^{-12}$ in solving (22) has been recorded and is shown in Fig. 7, left. The parameters $\alpha$ and $\beta$ arising in the theoretical upper bound (12) have been chosen to give a best fit. Apparently, the convergence behaviour is well captured by the theory—as long as the parameters $\alpha$ and $\beta$ are are chosen appropriately. When storing $L_\xi$, the results are similar. In contrast to storing $A_\xi^{-1}$, uniformly distributed errors in $[-1, 1]$ have almost no impact on

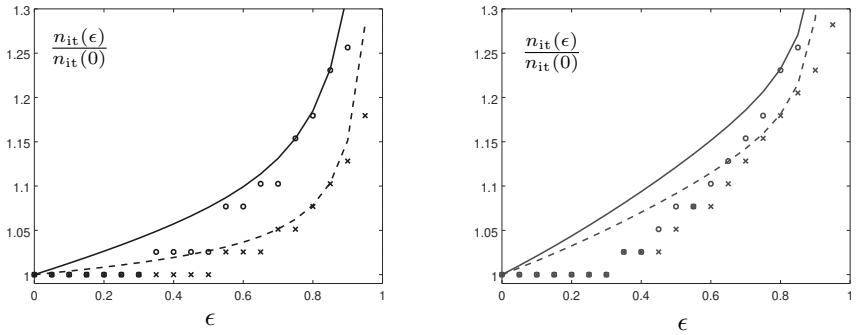


Figure 7: Relative increase $n_{\mathrm{it}}(\epsilon)/n_{\mathrm{it}}(0)$ of observed iteration numbers versus relative accuracy $\epsilon$ for $p = 3$ (circles) and $p = 4$ (crosses). Lines are the theoretically predicted increase factors with fitted parameters $\alpha$ and $\beta$. *Left:* storing $A_\xi^{-1}$, $s$ from (12) *Right:* storing $L_\xi$, $s$ from (15).

the iteration count, but uniformly distributed errors $(\delta_\xi)_{ij} \in [-(i-j+1)^{-1}, 0]$ have. The latter setup has been used to compare actual iteration numbers with the theoretical prediction (15) in Fig. 7, right, again when identifying the parameters $\alpha$ and $\beta$ from the data. As before, the theoretical bound provides a reasonable description of the observed iteration numbers. We note that the identified values for $\alpha$ and $\beta$ differ not only between different ansatz orders, which is to be expected, but also between $A_\xi^{-1}$ and $L_\xi$ storage. The reason can be that either the particular choice of inaccuracy does not represent the worst case covered by the theory or that the theoretical bounds are not sharp, or both.

From these results we may draw two conclusions. First, the increase factor $s$ for iteration count bounds given in (12) and (15) describes the observed increase in iteration counts reasonably well, but only if appropriate values for the parameters $\alpha$ and $\beta$ are known. They are, however, hard to obtain a priori. Using the always feasible choice $\alpha = \beta = 1$ for an adaptive selection of storage precision, similar to the adaptive scheme proposed in [1], will be overly pessimistic. Second, storage of $A_\xi^{-1}$ appears to be much more susceptible to the type of error arising from entry-wise quantization due to reduced precision representation. Though not explained by theory, the observation suggests that this error structure is far from worst case when storing Cholesky factors $L_\xi$. This makes the latter more attractive compared to storing $A_\xi^{-1}$ even beyond the smaller condition number entering the relative accuracy (16).

For real program executions, $A_\xi^{-1}$ and $L_\xi$ are stored using any of the reduced precision floating point or fixed point representations, and $\delta_\xi$ captures the quantization error as a consequence of this. Observed iteration counts $\tilde{n}_{\mathrm{it}}$ that exceed the reference iteration count $n_{\mathrm{it}} = 39$ are reported in Tab. 2 for different combinations of storage scheme, matrix entry representation, and block size $m^\square$ —blank fields correspond to executions that did not converge. Executions using the $\mathrm{FP}_{8,7}$ floating point representation all have the same per-block quantization, This does not hold for the fixed point representations, where quantization errors and PCG iterations tend to increase with the block size. Apparently, the convergence is unaffected for all but the very least accurate representations, in agreement with the theoretical results above.

**$A_\xi^{-1}$ (dense & packed)**

| $m^\square$ | $\mathrm{FP}_{8,7}$ $p=7$ | fixed point 8 bit 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 8 | $\tilde{n}_{\mathrm{it}}=41$ | 39 | 39 | 39 | 39 | 43 | 42 |
| 16 | | 39 | 39 | 39 | 39 | 52 | |
| 32 | | 40 | 39 | 39 | 40 | 48 | |
| 64 | | 40 | 40 | 40 | 42 | 50 | |
| 128 | | 42 | 42 | 43 | 50 | | |
| 256 | | 42 | 45 | 47 | | | |

Notes for (representation | $p$ | format):
($\mathrm{FP}_{11,52}$ | 7 | dense): out of memory
($\mathrm{FP}_{11,52}$ | 8 | dense, packed): out of memory
($\mathrm{FP}_{8,23}$ | 8 | dense, packed): out of memory
($\mathrm{FP}_{8,7}$ | 8 | dense): out of memory
($\mathrm{FP}_{8,7}$ | 8 | packed): no convergence
(fixed point 16 bit | 8 | dense): out of memory

**$L_\xi$ (packed)**

| $m^\square$ | fixed point 8 bit 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|
| 8 | 39 | 39 | 39 | 39 | 39 | 40 |
| 16 | 39 | 39 | 39 | 39 | 39 | 53 |
| 32 | 39 | 39 | 39 | 39 | 42 | |
| 64 | 39 | 40 | 42 | 48 | 63 | |
| 128 | 41 | 48 | 59 | 82 | | |
| 256 | 50 | 89 | | | | |

Notes for (representation | $p$):
($\mathrm{FP}_{11,52}$ | 8): out of memory
($\mathrm{FP}_{8,23}$ | 8): out of memory

**$L_\xi$ (dissection)**

| $m^\square$ | fixed point 8 bit 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|
| 8 | 39 | 39 | 39 | 39 | 39 | 39 |
| 16 | 39 | 39 | 39 | 39 | 39 | 40 |
| 32 | 39 | 39 | 39 | 39 | 42 | 53 |
| 64 | 39 | 39 | 41 | 47 | 64 | |
| 128 | 39 | 44 | 53 | 77 | | |
| 256 | 40 | 59 | | | | |

Notes for (representation | $p$):
($\mathrm{FP}_{11,52}$ | 8): out of memory
($\mathrm{FP}_{8,23}$ | 8): out of memory

Table 2: Number of PCG iterations $\tilde{n}_{\mathrm{it}}$ for the test problem setup and storage schemes $A_\xi^{-1}$ (dense and packed) and $L_\xi$ (packed and dissection) and ansatz orders $p$. In all cases, except the listed ones, $\tilde{n}_{\mathrm{it}} = n_{\mathrm{it}} = 39$. Executions that ran out of memory (192 GiB) are noted below the tables.

**Run times.** Figure 8 illustrates the patch smoother application time for storing either $A_\xi^{-1}$ (dense and packed) or $L_\xi$ (packed and dissection) and different ansatz orders $p$, floating point and fixed point representations $\mathrm{FP}_{R,P}$, and block sizes $m^\square$ for our test problem setup.

As the smoother application time is dominated by the execution time(s) of the BLAS level 2 operation(s) used, our observations from Sec. 4.1 can essentially be transferred. Up to ansatz order $p = 6$, all configurations can be compared amongst each other, and performance gains over the original program version storing $A_\xi^{-1}$ as dense matrix can be deduced (on our Skylake system):

1. the impact of the reduced floating point and fixed point representations

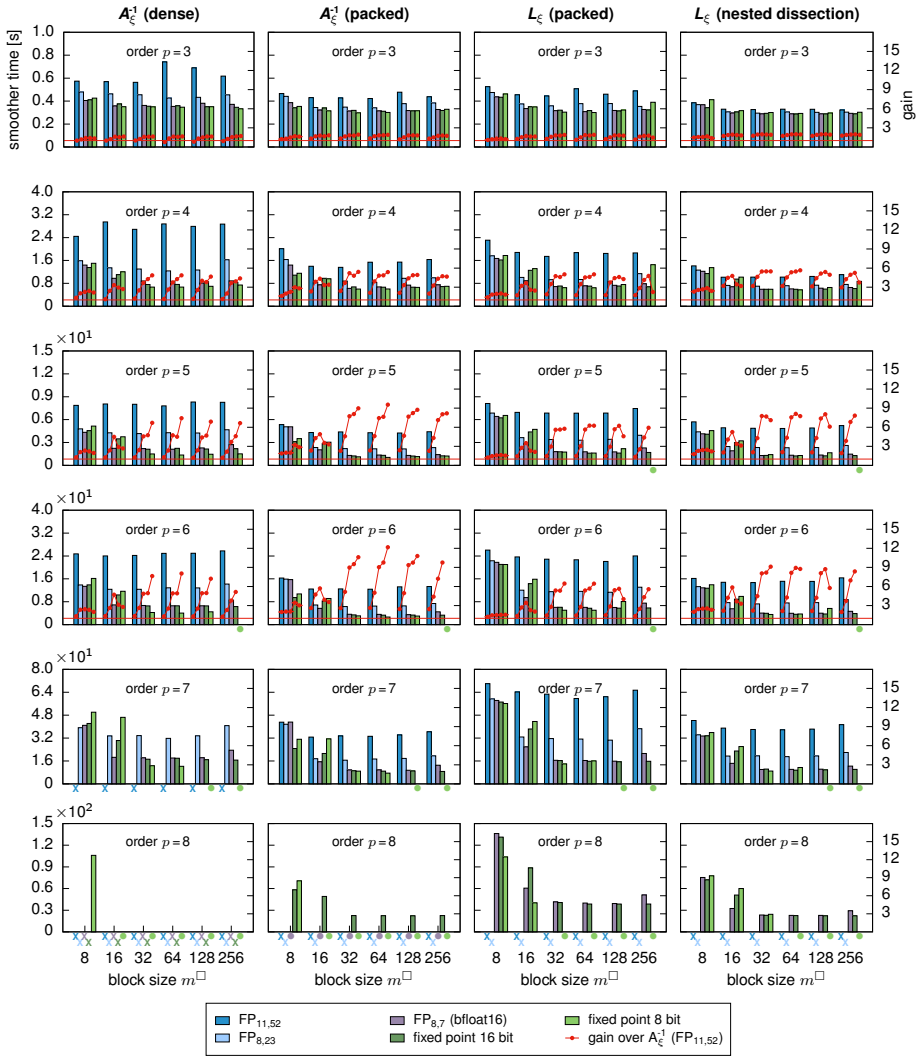


Figure 8: Patch smoother applications times in seconds for storing either $A_\xi^{-1}$ (dense and packed) or $L_\xi$ (packed and dissection) and different orders $p$, floating point and fixed point representations $\mathrm{FP}_{R,P}$, and block sizes $m^\square$ for our test problem setup. Performance gains over storing $A_\xi^{-1}$ (dense) in the original code base using the double type (and *gemv* for processing) are displayed as red dots. Crosses and bullets below the $x$-axis denote executions that ran out of memory (**x**) or did not converge (•), as noted in Tab. 2.

becomes significant only for ansatz order $p \geq 4$ and block sizes $m^\square \geq 32$.

2. the expected performance gains due to the reduced memory footprints are observed only for the 16-bit and 32-bit representations.

3. the 8-bit fixed point representation is in front only when storing $A_\xi^{-1}$ as dense matrix and with $m^\square \leq 128$. In all other cases, the computations seem to be less memory bound and the increased PCG iteration counts for

$m^{\square} \geq 32$ (see Tab. 2) additionally diminish the achievable performance gain.

Going beyond $p = 6$, the increased memory footprint prevents the full storage of $A_\xi^{-1}$ in double precision, such that no speedups over this scheme are reported in Fig. 8 for $p > 6$. Storing $A_\xi^{-1}$ in packed format or with reduced precision representations still fits into memory, such that run times can be reported.

Packed symmetric storage of $A_\xi^{-1}$ gains the performance over storing the dense matrix by halving the amount of data being processed—calls to *gemv* are replaced by calls to *spmv*. Consequently, the second column in Fig. 8 shows a factor two performance gain already for the $\mathrm{FP}_{11,52}$ floating point representation if $p \geq 4$ and $m^{\square} \geq 32$. Storing $L_\xi$ in packed format, however, cannot benefit from symmetry, and falls behind packed $A_\xi^{-1}$. The greater robustness of Cholesky factors with respect to inexact storage is visible only for $p = 8$.

For higher orders $p \geq 4$, the (single level and hence non-nested) dissection format cuts down the amount of nonzeros to be stored by almost a factor of two compared to packed storage of $L_\xi$, and is therefore on par with packed storage of $A_\xi^{-1}$. We expect that, in particular for higher orders, adding one or two levels more of nested dissection would improve the speedup further, even though the possible gains are limited by the moderate sparsity of $A_\xi$.

# 5 Conclusions

In preconditioned conjugate gradient solvers for higher order finite element discretization of elliptic equations, the preconditioner often dominates the run time. One effective preconditioner bounding the condition number independently of the order $p$ involves an overlapping Schwarz smoother, which is memory bandwidth bound. Reducing the storage size can be done by exploiting symmetry and block sparsity, or reduced precision storage, or both. Theoretical accuracy requirements for reduced precision have been derived using subspace correction theory, and show a greater robustness of storing Cholesky factors than storing inverses. Being worst case bounds, these results are far from sharp, though, and a better theoretical understanding of quantization error impact on convergence is required for an effective adaptive choice of precision.

Numerical experiments show that quite low precision, down to 16 bits per number, is often sufficient for undisturbed PCG convergence, and effectively reduces storage size and preconditioner execution time roughly by the expected factor. For even lower precision of eight bits per number, the computation is apparently no longer bandwidth-bound, such that the additional gain diminishes.

In particular, for low precision storage, fixed point representations turn out to be more accurate and preferable compared to floating point storage. The results also demonstrate that the overhead for decompression is not negligible, as can be seen in the unsatisfactory results for small block size $m^{\square}$.

The combination of sparse factorization and controlled low precision fixed point representation is a promising way to adapt overlapping Schwarz smoothers to modern computer architectures with growing imbalance of memory bandwidth and CPU performance.

**Acknowledgements.** During the preparation of the manuscript, our friend and coauthor Jakob Schneck passed away unexpectedly. We miss him very much.

Helpful discussions with Thomas Steinke are gratefully acknowledged.

This work has been supported by the German Ministry for Eduction and Research (BMBF) under project grants 01IH16005 (HighPerMeshes) and 01EC1408B (OVERLOAD-PrevOP), and has been partially funded through an IPCC grant by Intel Corporation.

# References

[1] H. Anzt, J. Dongarra, G. Flegar, N.J. Higham, and E.S. Quintana-Ortí. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience*, 0(0):e4460.

[2] H. Anzt, P. Luszczek, J. Dongarra, and V. Heuveline. GPU-accelerated asynchronous error correction for mixed precision iterative refinement. In C. Kaklamanis, T. Papatheodorou, and P.G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*. Springer.

[3] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Compu. Phys. Comm.*, 180(12):2526–2533, 2009.

[4] P. Benedusi, D. Hupp, P. Arbenz, and R. Krause. A parallel multigrid solver for time–periodic incompressible Navier–Stokes equations in 3D. In *Numerical Mathematics and Advanced Applications ENUMATH 2015*, pages 265–273. Springer, 2016.

[5] Ion Bică. *Iterative Substructuring Algorithms for the p-version Finite Element Method for Elliptic Problems*. PhD thesis, Courant Institute, New York University, 1997.

[6] J.H. Bramble, J.E. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.

[7] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys. Implications of reduced-precision computations in HPC: Performance, energy and error. In S. Bassini and M. Danelutto, editors, *Parallel Computing is Everywhere*, volume 32 of *Advances in Parallel Computing*, pages 297–306. IOS Press, 2018.

[8] T.A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[9] T.A. Davis, S. Rajamanickam, and W.M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 2006.

[10] P. Deuflhard and M. Weiser. *Adaptive numerical solution of PDEs*. de Gruyter, 2012.

[11] A. George and J.W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[12] L. Giraud, A. Haidar, and L.T. Watson. Mixed-precision preconditioners in parallel domain decomposition solvers. In U. Langer, M. Discacciati, D.E. Keyes, O.B. Widlund, and W. Zulehner, editors, *Domain Decomposition Methods in Science and Engineering XVII*, volume 60 of *Lecture Notes in Computational Science and Engineering*, pages 357–364. Springer, 2008.

[13] S. Götschel, C. von Tycowicz, K. Polthier, and M. Weiser. Reducing memory requirements in scientific computing and optimal control. In T. Carraro, M. Geiger, S. Körkel, and R. Rannacher, editors, *Multiple Shooting and Time Domain Decomposition Methods*, pages 263–287. Springer, 2015.

[14] S. Götschel, M. Weiser, and A. Schiela. Solving optimal control problems with the Kaskade 7 finite element toolbox. In A. Dedner, B. Flemisch, and R. Klöfkorn, editors, *Advances in DUNE*, pages 101–112. Springer, 2012.

[15] D. Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. PhD thesis, U Dortmund, 2010.

[16] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices, Part I: introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999.

[17] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Trans. Vis. Comp. Graphics*, 20(12):2674–2683, 2014.

[18] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[19] R. Pasquetti, L.F. Pavarino, F. Rapetti, and E Zampieri. Overlapping Schwarz preconditioners for Fekete spectral elements. In O.B. Widlund and D.E. Keyes, editors, *Domain Decomposition Methods in Science and Engineering XVI*, volume 55 of *Lecture Notes in Computational Science and Engineering*. Springer, 2007.

[20] L. Pavarino and S. Scacchi. Multilevel additive Schwarz preconditioners for the bidomain reaction-diffusion system. *SIAM J. Sci. Comput.*, 31(1):420–443, 2008.

[21] L.F. Pavarino. Additive Schwarz methods for the $p$-version finite element method. *Numer. Math.*, 66(1):493–515, 1994.

[22] J. Schöberl, J.M. Melenk, C. Pechstein, and S. Zaglmayr. Additive Schwarz preconditioning for $p$-version triangular and tetrahedral finite elements. *IMA J. Num. Anal.*, 28(1):1–24, 2008.

[23] H. Sundar, G. Stadler, and G. Biros. Comparison of multigrid algorithms for high-order continuous finite element discretizations. *Numerical Linear Algebra Appl.*, 22(4):664–680, 2015.

[24] E.L. Wilson. The static condensation algorithm. *Int. J. Numer. Meth. Engin.*, 8(1):198–203, 1974.

[25] J. Xu. Iterative methods by space decomposition and subspace correction. *SIAM Review*, 34(4):581–613, 1992.

[26] O.C. Zienkiewicz, R.L. Taylor, and J.Z. Zhu. *The Finite Element Method*. Elsevier, 2005.