Adaptive Precision in Block-Jacobi Preconditioning for Iterative Sparse Linear System Solvers

Anzt, Hartwig and Dongarra, Jack and Flegar, Goran and Higham, Nicholas J. and Quintana-Orti, Enrique S.

2017

MIMS EPrint: **2017.35**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Adaptive Precision in Block-Jacobi Preconditioning for Iterative Sparse Linear System Solvers

Hartwig Anzt[1,3*], Jack Dongarra[3,4,5], Goran Flegar[2], Nicholas J. Higham[5], Enrique S. Quintana-Ortí[2]

[1] *Karlsruhe Institute of Technology, Karlsruhe, Germany.* hartwig.anzt@kit.edu
[2] *Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), Castellón, Spain.* {flegar,quintana}@uji.es
[3] *Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee, USA.* dongarra@icl.utk.edu
[4] *Oak Ridge National Laboratory, USA*
[5] *School of Mathematics, University of Manchester, UK.* nick.higham@manchester.ac.uk

SUMMARY

We propose an adaptive scheme to reduce communication overhead caused by data movement by selectively storing the diagonal blocks of a block Jacobi preconditioner in different precision formats (half, single, or double). This specialized preconditioner can then be combined with any Krylov subspace method for the solution of sparse linear systems to perform all arithmetic in double precision. We assess the effects of the adaptive-precision preconditioner on the iteration count and data transfer cost of a preconditioned conjugate gradient solver. A preconditioned conjugate gradient method is, in general, a memory-bound algorithm, and therefore its execution time and energy consumption are largely dominated by the costs of accessing the problem's data in memory. Given this observation, we propose a model that quantifies the time and energy savings of our approach based on the assumption that these two costs depend linearly on the bit length of a floating point number. Furthermore, we use a number of test problems from the SuiteSparse matrix collection to estimate the potential benefits of the adaptive block-Jacobi preconditioning scheme.

KEY WORDS: Sparse linear systems; Krylov subspace methods; conjugate gradient (CG) method; Jacobi preconditioners; adaptive precision; communication reduction; energy efficiency

1. INTRODUCTION

Krylov subspace-based iterative methods for the solution of sparse linear systems typically benefit from the integration of a preconditioner that improves the conditioning of the linear system and, consequently, accelerates the convergence process [1].

A popular preconditioner is the Jacobi preconditioner and its block-Jacobi variants. Preconditioners of this class are based on simple (block-)diagonal scaling, which makes them highly parallel schemes suitable for fine-grained parallelism, and they have proven to provide a fair acceleration for many applications. For example, block-Jacobi preconditioners can efficiently efficiently exploit the massive hardware concurrency of graphics processing units (GPUs) [2, 3].

For virtually all current hardware technologies, the computational performance of preconditioned Krylov methods is limited by the memory bandwidth and depends heavily on the cost of memory access. Furthermore, for current architectures, data movement is not just a performance constraint but also a major source of energy consumption. Therefore, with highly parallel, high-performance computing (HPC) systems moving in the direction of an increasing floating point operations

*Correspondence to: Hartwig Anzt, Karlsruhe Institute of Technology, Steinbuch Centre for Computing, Germany. hartwig.anzt@kit.edu

(FLOP) per byte ratio, innovative techniques to reduce communication are critical for future applications [4, 5, 6, 7].

When a block-Jacobi preconditioner is combined with a simple Krylov iterative method—like the preconditioned conjugate gradient (PCG) method, which is suitable for the solution of sparse linear systems with a symmetric positive–definite (SPD) coefficient matrix [1]—a significant portion of the accesses to main memory is caused by the application of the preconditioner at each iteration. To decrease the costs of this stage, we analyze a version of the block-Jacobi preconditioner that selectively stores part of its data in low precision. This strategy reduces the data access volume during the application of the block-Jacobi preconditioner. We emphasize that, for a memory-bounded operation such as the PCG method, the time and energy savings of operating with reduced precision mostly come from the reduction of the volume of data being transferred, not from the increase in the single instruction, multiple data (SIMD) capacity associated with using reduced-precision arithmetic. Therefore, our solution aims to reduce the cost of communication due to the preconditioner application only. All other data (including the sparse matrix entries) as well as all arithmetic occurs in the conventional double precision. In more detail, our work makes the following contributions:

- We propose an adaptive preconditioner that *stores* the diagonal Jacobi blocks in the preconditioner using half, single, or double precision, depending on the conditioning and data range. In our scheme, the preconditioner blocks are retrieved from memory in the corresponding format and transformed into double precision once in the processor registers; all arithmetic operations are then performed at double precision level. As stated earlier, the entries for the sparse matrix and recurrence vectors for the conjugate gradient (CG) method (or any other Krylov subspace method) are maintained and retrieved in main memory using standard double precision.
- We investigate the impact that storing a block-Jacobi preconditioner in low precision exerts on the PCG convergence rate and the effectiveness of the adaptive precision block–Jacobi at maintaining the reference convergence rate.
- We develop a model that quantifies the runtime and energy savings based on the assumption that these costs depend linearly on the bit length of a floating point number.
- We use a set of test problems from the SuiteSparse matrix collection [8] to analyze the robustness of the adaptive preconditioning in a CG method, and to estimate the potential energy savings.

The use of mixed precision in preconditioned iterative solvers was previously explored with a primary focus on reducing the cost of arithmetic operations. In [9], Arioli and Duff show that, when using a lower-upper (LU) preconditioner computed in single precision within a flexible generalized minimal residual method (GMRES) based iterative solver (which enables one to use a non-constant preconditioning operator), backward stability at double precision can be preserved even for ill-conditioned systems. In [10], Carson and Higham provide a detailed error analysis of LU-based mixed refinement approaches for ill-conditioned systems. In [11], the same authors go as far as using half precision for computing an LU preconditioner that is used in the solution process of a GMRES solver that is part of a mixed precision iterative refinement process.

Our approach is fundamentally different. We do not aim to employ reduced precision in the generation or application of the preconditioner nor in any other arithmetical computation. Instead, we preserve full precision in all computations but store part of the preconditioner at a reduced precision. After reading the preconditioner stored at reduced precision, all data is converted to full precision before proceeding with the arithmetic operations in the actual preconditioner application. We argue that this approach has significantly higher potential for runtime- and energy savings than the previously proposed strategies for three reasons: (1) since the performance of sparse linear algebra algorithms is typically memory bound, the performance benefit obtained by reducing the data access volume is greater than the benefit obtained by reducing the cost of FLOPs; (2) since the energy cost of data access is more than an order of magnitude greater than that of arithmetic operations [12], more resources can be saved by reducing data accesses; and (3) running the

preconditioner application at reduced precision results in a preconditioning operator not preserving orthogonality in double precision, implying that previously orthogonal Krylov vectors may not be orthogonal after the preconditioner application. To account for this situation, flexible variants that introduce an additional orthogonalization step are required to preserve convergence [13]. Performing arithmetic operations in the distinct preconditioner applications in full precision (even though the preconditioner data is stored at reduced precision) preserves the orthogonality of the Krylov subspace and removes the burden of expensive reorthogonalization.

Section 2 provides a more detailed discussion of the need for reorthogonalization when using mixed-precision preconditioning in flexible Krylov methods. This discussion includes a detailed description of the CG solver that we later employ as a Krylov solver for testing our adaptive precision block-Jacobi preconditioner. A brief recap/overview of block-Jacobi preconditioning is provided in Section 3. In Section 4, we introduce the concept of adaptive precision preconditioning, and we introduce the evaluation criteria for selecting the storage format of the distinct diagonal blocks. Rounding error analysis to support the criteria is given in Section 5. We report the experimental results in Section 6, which includes an analysis of "reckless" precision reduction in block-Jacobi preconditioning, the assessment of the evaluation criteria, and an energy consumption model that quantifies the savings owed to adaptive precision preconditioning. We summarize our findings in Section 7 and provide details about the path forward for this research.

2. REDUCED PRECISION PRECONDITIONING IN THE PCG METHOD

2.1. *Brief review*

Figure 1 shows the PCG method for the solution of the linear system $Ax = b$; where the coefficient matrix, $A \in \mathbb{R}^{n\times n}$, is SPD and sparse with n_z nonzero entries; $b \in \mathbb{R}^n$ is the right-hand side; and $x \in \mathbb{R}^n$ is the sought-after solution. The most challenging operations in this algorithm are the computation of the preconditioner (before the iteration commences), the computation of the sparse matrix-vector product (SPMV) (at each iteration), and the preconditioner application (at each iteration). The remaining operations are scalar computations or simple vector kernels like the dot product (DOT) and AXPY-type vector updates [14].

$A \rightarrow M$
Initialize $x_0, p_0, r_0 := b - Ax_0, \tau_0 :=\| r_0 \|_2, \gamma_0$
$k := 0$
while $(\tau_k > \tau_{\max})$
  $q_{k+1} := Ap_k$
  $\eta_k := p_k^T q_{k+1}$
  $\alpha_k := \gamma_k/\eta_k$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k q_{k+1}$
  $\tau_{k+1} :=\| r_{k+1} \|_2$
  $z_{k+1} := M^{-1} r_{k+1}$
  $\gamma_{k+1} := r_{k+1}^T z_{k+1}$
  $\beta_{k+1} := \gamma_{k+1}/\gamma_k$
  $p_{k+1} := z_{k+1} + \beta_{k+1} p_k$
  $k := k + 1$
endwhile

Figure 1. Mathematical formulation of the PCG method. Here, $\tau_{\max}$ is the relative residual stopping criterion.

In the PCG method, the DOT operations present a one-to-one ratio of FLOPs to memory accesses (MEMOPS), and the AXPY-type operations present a two-to-three ratio of FLOPs to MEMOPS, which clearly identifies these operations as memory-bounded kernels. For simplicity, moving forward we make no distinction between the cost of reading a number and the cost of writing

a number. Assuming the sparse matrix is stored in compressed sparse row (CSR) format [1]—and is using 64 bits for double precision numbers/values (fp64) and 32 bits for integers/indices (int32)—the ratio of FLOPs:MEMOPS for SPMV is $2n_z/((n+n_z)\cdot \text{fp64} + (n+n_z)\cdot \text{int32})$. As a consequence, this operation is also memory bounded. An analysis of the operations using the preconditioner is provided later in this section.

2.2. *Flexible CG*

Using a reduced-precision preconditioner (i.e., 32-bit or 16-bit arithmetic) instead of "full" 64-bit, double-precision arithmetic requires a careful consideration of the numerical effects. The PCG method presented in Figure 1 assumes that the preconditioner is a constant operator acting on the input vector, $r = r_{k+1}$, as $z = z_{k+1} := M^{-1}r$ [1]. In this case, $r_k^T z_{k+1} = 0$, that is to say, the orthogonality with respect to the previous residual is preserved. Strictly speaking, even when using double precision, the preconditioner application introduces some rounding error so that the computed operator satisfies $z = M^{-1}r + \mathcal{O}(\varepsilon_d)$, where ε_d stands for fp64 machine precision. Hence, a preconditioner in double precision can also have an impact on the orthogonality. However, as the effects are in the order of the approximation accuracy, the non-consistency of the preconditioning operator is typically disregarded in practice.

In contrast, when applying a preconditioner in less than double precision, this issue becomes more relevant, because the rounding error now grows to $z = M^{-1}r + \mathcal{O}(\varepsilon_r)$, where ε_r is the machine precision of the reduced format. As a result, the orthogonality error increases to ε_r, which becomes relevant if convergence beyond ε_r is desired.

A straightforward workaround is to introduce an additional orthogonalization step to account for the loss in orthogonality. Concretely, replacing the Fletcher-Reeves formula from Figure 1,

$$\beta_k := \gamma_{k+1}/\gamma_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k}, \tag{1}$$

with the Polak-Ribière formula,

$$\beta_k := \frac{(r_{k+1} - r_k)^T z_{k+1}}{r_k^T z_k}, \tag{2}$$

naturally accounts for z_{k+1} losing orthogonality with respect to the previous residual, r_k [13]. Compared with the original formulation of the CG method, this orthogonality-preserving "flexible CG" (FCG) [13] incurs an overhead that corresponds to keeping the last residual vector in memory and computing an additional vector operation and DOT product. The benefits are that the iterative method can handle a flexible (non-constant) preconditioner [15], which is needed when applying a preconditioner in reduced precision.

Obviously, with a constant preconditioner, $r_k^T z_{k+1} = 0$, i.e. both formulas (1) and (2) are identical. For $r_k^T z_{k+1} \neq 0$, the Polak-Ribière formula specifies a locally optimal search direction, which means that the convergence rate of this method will not be slower than that of a locally optimal steepest descent method [16]. We complement the preconditioned CG method, based on the Fletcher-Reeves formula shown in Figure 2, with the flexible conjugate gradient (FCG) method based on the Polak-Ribière formula in Figure 3. The two codes differ only in lines 6–8 (computation of `gamma_new` and additional recurrence for vector `t`), which results in $7n$ additional memory accesses. A faster preconditioner application (i.e., using reduced-precision arithmetic operations in the actual preconditioner application) could barely compensate for this overhead.

In our approach, we store the preconditioner at reduced precision, but we convert the data to double precision right after reading it from memory and before invoking the arithmetic computations of the preconditioner application. Hence, although stored at a reduced precision, the preconditioner itself remains constant across all iterations. This strategy does introduce some overhead in terms of converting the preconditioner data to double precision and using double precision in all arithmetic operations, but it comes with the benefit of using the Fletcher-Reeves formula (1) for the orthogonalization step, which results in the more attractive (in terms of memory) standard PCG solver.

```
% PCG method
% Compute preconditioner A -> M
% Initialize x, p, r = b - A * x, tau = r' * r, gamma_old
while (tau > tau_max)
    z           = M \ r;                % = M^{-1} r, apply preconditioner

    gamma_new   = r' * z;               % DOT product

    beta        = gamma_new / gamma_old; % scalar operation
    p           = z + beta * p;         % AXPY-type
    q           = A * p;                % SpMV
    eta         = p' * q;               % DOT product
    alpha       = gamma_new / eta;      % scalar operation
    gamma_old   = gamma_new;            % scalar operation
    x           = x + alpha * p;        % AXPY
    r           = r - alpha * q;        % AXPY
    tau         = r' * r;               % = ||r||_2,  DOT product
end
```

Figure 2. Algorithmic formulation (in MATLAB) of the PCG method. For a problem of size n containing n_z nonzero elements in the system matrix stored in CSR format, ignoring the preconditioner application, each PCG iteration requires $(14n + n_z) \cdot$ fp64 $+ (n_z + n) \cdot$ int32 memory transactions.

```
% Flexible CG method
% Compute preconditioner A -> M
% Initialize x, p, r = b - A * x, r_old, tau = r' * r, gamma_old
while (tau > tau_max)
    z           = M \ r;                % = M^{-1} r, apply preconditioner
    gamma_new   = r' * z;               % DOT product
    t           = r - r_old;            % AXPY-type
    gamma_t     = t' * z;               % DOT product
    r_old       = r;                    % COPY
    beta        = gamma_t / gamma_old;  % scalar operation
    p           = z + beta * p;         % AXPY-type
    q           = A * p;                % SpMV
    eta         = p' * q;               % DOT product
    alpha       = gamma_new / eta;      % scalar operation
    gamma_old   = gamma_new;            % scalar operation
    x           = x + alpha * p;        % AXPY
    r           = r - alpha * q;        % AXPY
    tau         = r' * r;               % = ||r||_2,  DOT product
end
```

Figure 3. Algorithmic formulation (in MATLAB) of the FCG method. For a problem of size n containing n_z nonzero elements in the system matrix stored in CSR format, ignoring the preconditioner application, each FCG iteration requires $(21n + n_z) \cdot$ fp64 $+ (n_z + n) \cdot$ int32 memory transactions.

3. BLOCK-JACOBI PRECONDITIONING

The Jacobi method splits the coefficient matrix as $A = L + D + U$, with a diagonal matrix $D = (\{a_{ii}\})$, a lower triangular factor $L = (\{a_{ij} \ : \ i > j\})$, and an upper triangular factor $U = (\{a_{ij} \ : \ i < j\})$. The block-Jacobi variant is an extension that gathers the diagonal blocks of A into $D = (D_1, D_2, \ldots, D_N)$, with $D_i \in \mathbb{R}^{m_i \times m_i}$, $i = 1, 2, \ldots, N$, and $n = \sum_{i=1}^{N} m_i$. The remaining elements of A are then partitioned into matrices L and U such that L contains the elements below the diagonal blocks in D, while U contains those above them [2]. The block-Jacobi method is well defined if all diagonal blocks are nonsingular. The resulting preconditioner, $M = D$, is particularly effective if the blocks succeed in reflecting the nonzero structure of the coefficient matrix, A. Fortunately, this is the case for many linear systems that, for example, exhibit some inherent block

structure because they arise from a finite element discretization of a partial differential equation (PDE) [2].

There are several strategies to integrate a block-Jacobi preconditioner into an iterative solver like CG. In this work, we adopt an approach that explicitly computes the block-inverse matrix, $D^{-1} = (D_1^{-1}, D_2^{-1}, \ldots, D_N^{-1}) = (E_1, E_2, \ldots, E_N)$, before the iterative solution process commences, and then applies the preconditioner in terms of a dense matrix-vector multiplication (GEMV) per inverse block E_i. Note that GEMV is still a memory-bounded kernel, independent of the block size. In practice, this strategy shows numerical stability similar to the conventional alternative that computes the LU factorization (with partial pivoting) [17] of each block ($D_i = L_i U_i$) and then applies the preconditioner using two triangular solvers (per factorized block). By comparison, the GEMV kernel is highly parallel, while the triangular solves offer only limited parallelism.

4. ADAPTIVE-PRECISION BLOCK-JACOBI PRECONDITIONING

The main goal of this work is to assess the potential benefits of a specialized version of a block-Jacobi preconditioner that selectively stores part of its data at low precision—a technique that reduces the memory access volume during the application of a block-Jacobi preconditioner. Concretely, we employ three precision formats: (1) 16-bit, half-precision arithmetic (fp16); (2) 32-bit, single-precision arithmetic (fp32); and (3) 64-bit, (full) double-precision arithmetic (fp64). The fp32 and fp64 roughly correspond to the two IEEE standards that are currently supported by practically all commodity processors used in everything from desktop systems to high-performance servers. On the other hand, fp16 has only recently received considerable attention because of its usefulness in deep learning applications, and hardware support for this format is now included in the most recent many-core architectures from NVIDIA.

For our experiments, we use a PCG Krylov solver to expose the effects of storing parts of a block-inverse preconditioner at a reduced precision. Before we introduce our preconditioning scheme and the strategy for selecting the appropriate storage format, we note that, for the type of systems that can be tackled using a CG method, the diagonal blocks of A in the preconditioner D are all symmetric. Therefore, a significant amount of storage (and communication cost) can already be saved by explicitly storing only the lower or upper triangular part of each block. We also recognize that some computational cost can be saved by exploiting the symmetry and positive definiteness information of these diagonal blocks. However, as these two cost-saving techniques are orthogonal to those we propose, we refrain from mixing the distinct strategies.

In general, the design of a block-Jacobi preconditioner with adaptive precision is based on the following observations.

1. In the preconditioner matrix, D, each one of the blocks, D_i, is independent.
2. Except for cases where the iterative solver converges quickly, the overhead incurred by determining an appropriate storage format for the preconditioner (before the iteration commences) is irrelevant.
3. The application of each block, D_i, (i.e., multiplication with the inverse block, E_i) should be done with care to guarantee "enough" precision in the result. As we will show in Section 5, the accuracy of this application is largely determined by the condition number of D_i with respect to inversion, denoted hereafter as $\kappa_1(D_i) = \|D_i\|_1 \|D_i^{-1}\|_1 = \|D_i\|_1 \|E_i\|_1$, [17].

Armed with these observations, we propose the following adaptive-precision block-Jacobi preconditioner:

1. Before the iteration commences, the inverse of each block, D_i, is computed explicitly using fp64: $D_i \rightarrow E_i$. We note that even if D_i is sparse, its inverse, E_i, is likely a dense matrix. For this reason, we store the inverse, E_i, following the conventional column-major order using $m_i \times m_i$ real numbers.
2. At the same stage (i.e., before the iteration), we compute $\kappa_1(D_i) = \kappa_1(E_i) = \|D_i\|_1 \|E_i\|_1$ and we note that, given E_i is explicitly available, computing $\kappa_1(D_i)$ is straightforward and inexpensive compared with the inversion of the block.

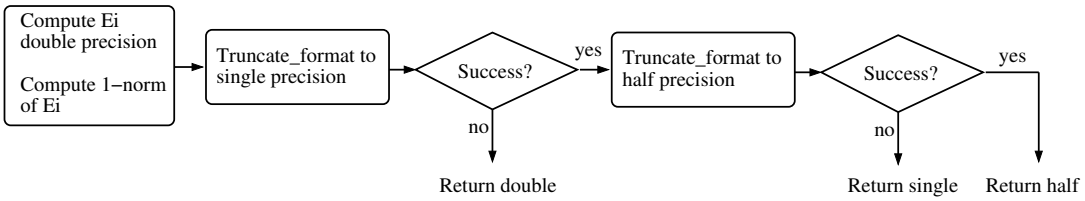

Figure 4. Control flow for deciding whether or not to select a reduced format.

3. In principle, we store E_i, which was computed in fp64, in the format determined by its condition number—truncating the entries of the block if necessary—as:

$$\begin{cases} \text{fp16} & \text{if } \tau_h^L < \kappa_1(D_i) \leq \tau_h^U, \\ \text{fp32} & \text{if } \tau_s^L < \kappa_1(D_i) \leq \tau_s^U, \text{ and} \\ \text{fp64} & \text{otherwise,} \end{cases} \tag{3}$$

with $\tau_h^L = 0$ and $\tau_h^U = \tau_s^L$. As we will discuss in Section 5, the values for the bounds τ_h^U and τ_s^U are selected by taking into account the unit roundoff for each format: $u_h \approx 4.88e-04$ for half precision, $u_s \approx 5.96e-08$ for single precision, and $u_d \approx 1.11e-16$ for double precision.
4. During the iteration, we recover the block E_i stored in the corresponding format in memory, transform its entries to fp64 once in the processor registers, and apply it in terms of a fp64 GEMV to the appropriate entries of r_{k+1} to produce those of z_{k+1}. This is a memory-bounded operation, and, therefore, its cost is dominated by the overhead of recovering the data for the preconditioner matrix and the vectors from memory (i.e., MEMOPS). Thus, we can expect that in practice the FLOPs will be completely "amortized" (i.e., overlapped) with the data transfers.

The truncation procedure for converting fp64 data to a reduced-precision format requires some care to deal with overflows/underflows and their consequences, as described below.

- The truncation of a "large" (in magnitude) value in E_i, represented in fp64, can produce an overflow because the number is too large to be represented in the reduced format, resulting in an "Inf" value in that format. In those cases, we can either discard the use of the reduced format for the complete block E_i or replace the truncated value with the largest number (in magnitude) representable in that format (e.g., for positive values, $65,504$ in fp16 and about $3.40e+38$ in fp32).
- Conversely, the truncation of a "small" (in magnitude) value, in fp64, may yield an underflow that returns a value that is zero. This can turn a nonsingular matrix E_i into a singular matrix. For example, if all entries of E_i are below the minimum representable number in the reduced format, the result of truncation will produce a block that comprises only zeros, and the preconditioned solver will not converge. This could be mitigated to some extent by scaling all the values of the block. Furthermore, even if some of the entries are nonzero the truncated representation of E_i may still become ill-conditioned, thereby causing numerical difficulties for the convergence. In order to avoid this issue, we propose checking the condition number of the truncated representation and not using the corresponding reduced precision if it is above the relevant threshold, τ_κ.

Figure 4 summarizes the global precision selection process, and the pseudocode in Figure 5 provides a practical implementation of the truncation procedure and the various thresholds—taking E_i and $\kappa_1(E_i)$ as inputs. The routine given in the pseudocode, `force_reduction`, simply truncates the fp64 block to a reduced format. The rest of the code uses several metrics to determine whether the use of the reduced format is safe.

```
function [Ei, success] = truncate_format(Ei, Di_cond_num1,...
                                         tau_r_L, tau_r_U, tau_k)
%
% Inputs : mi x mi dense inverse block Ei;
%          condition number of Di (and Ei) in Di_cond_num1; and
%          thresholds to determine use of reduced format:
%            - tau_r_L and tau_r_U (with tau_r=tau_h or tau_s), and
%            - tau_k
% Output : mi x mi dense inverse block Ei
%          overwritten with the reduced format if applicable
%
success = 0;                                % FALSE
if (tau_r_L < Di_cond_num1) & (Di_cond_num1 <= tau_r_U)
  Ei_reduced      = force_reduction(Ei); % Truncate to reduced format
  Ei_reduced_nrm1 = norm(Ei_reduced,1);
  if (Ei_reduced_nrm1 > 0.0)                % Ei contains nonzero entries
    % Compute the condition number of truncated block via explicit
    % inverse: easier to implement on GPU than SVD
    Ei_reduced_cond_num1 = Ei_reduced_nrm1 * norm(inv(Ei_reduced),1);
    if (Ei_reduced_cond_num1 < tau_k)     % Ei is not (close to) singular
      Ei                 = Ei_reduced;
      success = 1;                          %TRUE
    end
  end
end
%
return;
```

Figure 5. Details of the procedure for deciding whether or not to select a reduced format.

5. ROUNDING ERROR ANALYSIS

As previously elaborated, we invert the diagonal blocks explicitly using double precision, e.g. via (batched) Gauss-Jordan Elimination [2]. Let $E_i = D_i^{-1}$ be the inverse of block i computed in double precision arithmetic with unit roundoff u_d. By storing the inverse in reduced precision ($\widehat{E}_i$) with unit roundoff u, we introduce the error ΔE_i and get [18], [19, secs. 14.3, 14.4]

$$\widehat{E}_i = E_i + \Delta E_i, \qquad \|\Delta E_i\|_1 \leq c_{m_i} \kappa_1(D_i) \|\widehat{E}_i\|_1 u_d + u \|\widehat{E}_i\|_1, \tag{4}$$

for some constant c_{m_i}. For the vector segments in z_i and r_i corresponding to the diagonal block i, the subsequent multiplication in double precision results in [19, sec. 3.5]

$$\widehat{z}_i = fl(\widehat{E}_i r_i) = \widehat{E}_i r_i + \Delta z_i, \qquad \|\Delta z_i\|_1 \leq c'_{m_i} u_d \|\widehat{E}_i\|_1 \|r_i\|_1. \tag{5}$$

Hence

$$\widehat{z}_i = (E_i + \Delta E_i) r_i + \Delta z_i = E_i r_i + \overline{\Delta z}_i, \tag{6}$$

where combining (4) and (5) gives

$$\begin{aligned} \|\overline{\Delta z}_i\|_1 &= \|\Delta E_i r_i + \Delta z_i\|_1 \\ &\leq c_{m_i} \kappa_1(D_i) \|\widehat{E}_i\|_1 \|r_i\|_1 u_d + u \|\widehat{E}_i\|_1 \|r_i\|_1 + c'_{m_i} u_d \|\widehat{E}_i\|_1 \|r_i\|_1 \\ &= \big(c_{m_i} \kappa_1(D_i) u_d + u + c'_{m_i} u_d\big) \|\widehat{E}_i\|_1 \|r_i\|_1. \end{aligned} \tag{7}$$

We may assume that the constant term $c'_{m_i} u_d$ becomes negligible when storing the diagonal block in the reduced precision format with unit roundoff $u \gg u_d$. With this assumption,

$$\|\overline{\Delta z}_i\|_1 \leq c_{m_i} \big(\kappa_1(D_i) u_d + u\big) \|\widehat{E}_i\|_1 \|r_i\|_1. \tag{8}$$

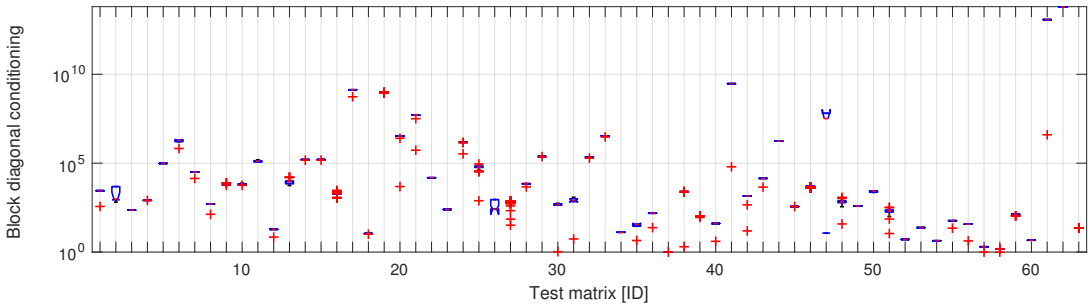

Figure 6. Boxplot for the distribution of the condition numbers of the diagonal blocks ($\kappa_1(D_i)$) using supervariable agglomeration with the block size set to 24. For each matrix, the blue central box shows where most of the condition numbers are located, the red crosses indicate outliers.

Noting that $r_i = E_i^{-1} z_i = D_i z_i$, this bound yields

$$\begin{aligned} \|\overline{\Delta z_i}\|_1 &\leq c_{m_i}\big(\kappa_1(D_i)u_d + u\big)\|\widehat{E}_i\|_1 \|D_i\|_1 \|z_i\|_1 \\ &\approx c_{m_i}\big(\kappa_1(D_i)u_d + u\big)\kappa_1(D_i)\|z_i\|_1, \end{aligned} \tag{9}$$

so that

$$\frac{\|\overline{\Delta z_i}\|_1}{\|z_i\|_1} \leq c_{m_i}\big(\kappa_1(D_i)u_d + u\big)\kappa_1(D_i). \tag{10}$$

As expected, the relative error depends on the conditioning of the diagonal block D_i. With the unit roundoff being a format-specific constant ($u_h \approx 4.88e-04$ for half precision, $u_s \approx 5.96e-08$ for single precision, and $u_d \approx 1.11e-16$ for double precision), (10) provides bounds for the relative error.

Recalling that we are within a preconditioner framework, by ignoring all entries outside the block-diagonal in the inversion process we may have already introduced a significant error. In fact, experiments reveal that preconditioners based on block-Jacobi often come with an error as large as $1.0e-2$ to $1.0e-1$. This makes it reasonable to allow for similar errors in (10), which yields the bounds for the condition numbers that are allowed in the respective formats. In the experimental section we use the bounds $\tau_h^U = \tau_s^L := 1.0e+2$, and $\tau_s^U := 1.0e+6$.

6. EXPERIMENTAL ANALYSIS

6.1. *Experimental framework*

In this section, we assess the potential benefits of the adaptive precision block–Jacobi preconditioner with a collection of experiments performed in GNU Octave version 3.8.1. We implement the PCG method according to [13] (Figure 2) with an integrated block-Jacobi preconditioner that performs an explicit inversion of the diagonal blocks. We apply supervariable agglomeration to optimize the block diagonal structure of the block-Jacobi preconditioner for the specific problems used here [20]. This procedure aims to identify and capture the block structure of the matrix in the Jacobi blocks of the preconditioner, thereby accumulating multiple blocks into a larger superstructure with the upper bound of the blocksize set to 24.

For the evaluation, we consider a subset comprised of 63 SPD test problems of small to moderate dimension from the SuiteSparse matrix collection [8]. We list the matrices along with some key characteristics in Table I.

In the adaptive precision preconditioner, we use the evaluation strategy shown in Figures 4 and 5 to determine the precision at which the individual diagonal blocks should be stored. According to the heuristics presented in Section 5, we set $\tau_h^L := 0$, $\tau_h^U = \tau_s^L := 1.0e+2$, $\tau_s^U := 1.0e+6$; and

ID	Matrix	# rows	# nonzeros	cond. number	#PCG iterations			
					double	single	half	adaptive
1	1138_bus	1,138	4,054	1.2100e+07	784	778	–	782
2	494_bus	494	1,666	3.8900e+06	269	269	271	269
3	662_bus	662	2,474	8.2100e+05	179	179	179	179
4	685_bus	685	3,249	4.0500e+05	171	171	–	172
5	bcsstk01	48	400	1.6000e+06	35	34	–	34
6	bcsstk03	112	640	6.2700e+06	46	48	–	48
7	bcsstk04	132	3,648	5.5500e+06	72	72	–	72
8	bcsstk05	153	2,423	3.5300e+04	95	95	–	95
9	bcsstk06	420	7,860	1.1900e+07	255	254	–	254
10	bcsstk07	420	7,860	1.1900e+07	255	254	–	254
11	bcsstk08	1,074	12,960	2.3200e+06	231	231	–	231
12	bcsstk09	1,083	18,437	3.6000e+03	325	325	–	325
13	bcsstk10	1,086	22,070	1.3200e+06	517	517	–	517
14	bcsstk11	1,473	34,241	4.2100e+06	768	764	–	764
15	bcsstk12	1,473	34,241	2.9000e+06	768	764	–	764
16	bcsstk13	2,003	83,883	5.6400e+08	1,639	1,631	–	1,444
17	bcsstk14	1,806	63,454	1.3100e+10	276	276	–	276
18	bcsstk15	3,948	117,816	1.9800e+07	585	584	–	583
19	bcsstk16	4,884	290,378	7.0100e+09	263	261	–	263
20	bcsstk19	817	6,853	5.8600e+10	1,775	1,773	–	1,768
21	bcsstk20	485	3,135	7.4800e+12	2,125	2,113	–	2,114
22	bcsstk21	3,600	26,600	2.6000e+06	565	565	–	565
23	bcsstk22	138	696	2.7600e+04	75	75	–	75
24	bcsstk24	3,562	159,910	7.1800e+10	2,505	2,630	–	2,336
25	bcsstk26	1,922	30,336	8.0800e+06	1,979	1,957	–	1,957
26	bcsstk27	1,224	56,126	1.4900e+04	213	213	–	213
27	bcsstk28	4,410	219,024	6.2800e+09	2,182	2,115	–	2,115
28	bcsstm07	420	7,252	1.3400e+04	46	46	46	46
29	bcsstm12	1,473	19,659	8.8800e+05	26	26	**1,220**	26
30	lund_a	147	2,449	9.8900e+05	89	90	–	90
31	lund_b	147	2,441	6.0300e+04	47	47	48	47
32	nos1	237	1,017	7.5900e+06	157	165	–	165
33	nos2	957	4,137	1.8300e+09	2,418	2,409	–	2,409
34	nos3	960	15,844	7.3500e+04	137	137	137	137
35	nos4	100	594	2.7000e+03	46	46	47	47
36	nos5	468	5,172	3.5900e+03	235	235	–	235
37	nos6	675	3,255	8.0000e+06	77	77	–	77
38	nos7	729	4,617	4.0000e+09	68	68	–	68
39	plat1919	1,919	32,399	2.2200e+18	4,117	4,049	3,772	4,081
40	plat362	362	5,786	7.0800e+11	982	**1,112**	1,115	1,095
41	mhdb416	416	2,312	5.0500e+09	19	19	–	19
42	bcsstk34	588	21,418	2.6700e+04	185	185	–	185
43	msc00726	726	34,518	8.5500e+05	160	160	–	160
44	msc01050	1,050	26,198	9.0000e+15	1,594	1,593	–	1,593
45	msc01440	1,440	44,998	7.0000e+06	929	928	–	928
46	msc04515	4,515	97,707	4.7800e+05	2,348	2,349	–	2,349
47	ex5	27	279	1.3200e+08	10	**25**	–	10
48	nasa1824	1,824	39,208	2.3100e+05	896	896	–	896
49	nasa2146	2,146	72,250	2.8100e+03	352	353	–	353
50	nasa2910	2,910	174,296	1.3000e+06	1,369	1,369	–	1,369
51	nasa4704	4,704	104,756	6.4500e+06	4,171	4,123	–	4,123
52	mesh1e1	48	306	8.2000e+00	14	14	14	14
53	mesh1em1	48	306	3.4000e+01	23	23	23	23
54	mesh1em6	48	306	8.8500e+00	14	14	15	15
55	mesh2e1	306	2,018	4.0700e+02	79	79	83	83
56	mesh2em5	306	2,018	2.7900e+02	77	77	81	75
57	mesh3e1	289	1,377	9.0000e+00	18	18	18	18
58	mesh3em5	289	1,377	5.0000e+00	17	17	17	17
59	sts4098	4,098	72,356	3.5600e+07	342	342	–	340
60	Chem97ZtZ	2,541	7,361	3.2900e+02	30	30	30	30
61	mhd3200b	3,200	18,316	2.0200e+13	17	17	–	17
62	mhd4800b	4,800	27,520	1.0300e+14	16	16	–	16
63	plbuckle	1,282	30,644	2.9200e+05	260	260	–	260

Table I. Left: Test matrices along with key properties. Right: Iteration count of the PCG method with the preconditioner stored in double, single, half, or adaptive precision. The "–" symbol indicates cases where the iterative solver did not reach the relative residual threshold $\tau_{\max} = 1.0e-9$ after 5,000 iterations.

$\tau_\kappa := 1.0e-3/u_d$; see also (3). Specifying an upper block size bound of 24 in the supervariable agglomeration, we show in Figure 6 the condition number distribution of the blocks for each test matrix. These condition numbers are one of the metrics considered when selecting the storage format in the adaptive precision block–Jacobi preconditioner.

Using Octave, we emulate the procedure for truncation of fp64 values to reduced precision formats (`force_reduction` shown in Figure 5) as follows. First, we transform the full precision value to a text string and then truncate that string to keep only the two most significant decimal digits for fp16 and the seven most significant decimal digits for fp32. This is a rough approximation of the precision level that can be maintained with the bits dedicated to the mantissa in the IEEE standards for fp16/fp32. To emulate overflow, we set values exceeding the data range of the low precision format to the largest representable number in the target format, $R_{\max}$, which is $R_{\max} = 65,504$ for fp16 and $R_{\max} = 3.40e+38$ for fp32. We preserve the sign in this truncation process. To emulate underflow, values that are smaller than the minimum value that can represented in the low precision format, $R_{\min}$, are set to zero, which is $R_{\min} = 6.10e-5$ for fp16 and $R_{\min} = 1.17e-38$ for fp32. We stop the PCG iterations once the relative residual has dropped below the threshold $\tau_{\max} := 1.0e-9$. We allow for, at most, 5,000 PCG iterations.

6.2. *Reduced-precision preconditioning*

In the first experiment, we investigate how a reckless/adaptive reduction of the precision for the representation of the block-Jacobi preconditioner impacts the convergence rate of the PCG iterative solver. By recklessly reducing the precision format used for storing the block-diagonal inverse, essential information may be lost, which slows down the convergence of the iterative solver. In the worst case, the diagonal blocks may become singular, or the entries may fall outside of the data range that can be represented in the chosen precision; both cases would likely result in the algorithm's breakdown. We emphasize that the distinct preconditioners only differ in the format that is leveraged to store the block inverse. Conversely, the problem-specific diagonal block pattern is not affected, and all computations are realized in fp64.

The three leftmost columns in the right part of Table I report the iterations required for convergence of the PCG method when storing the block-inverse preconditioner in fp64, fp32, or fp16. We observe that storing the preconditioner in fp32 usually has only a mild impact on the preconditioner quality. In most cases, the PCG iteration count matches the one where the preconditioner is stored in fp64. In a few cases, the PCG converges even faster when storing the preconditioner in fp32. Conversely, if the preconditioner is stored in fp16, the PCG does not converge in most cases. Therefore, fp16 storage cannot be recommended as the default choice. In the right-most column of Table I, we report the iteration count for the PCG method preconditioned with adaptive precision block–Jacobi. We observe that, except for some noise, the adaptive precision block–Jacobi preserves the quality of the preconditioner and the convergence rate of the fp64 solver. Figure 7 shows that most of the time the adaptively chosen precision is single or half precision, with relatively few instances o of double.

6.3. *Energy model*

Having validated that the adaptive precision block–Jacobi preconditioner preserves the convergence rate of the iterative solver, we next quantify the advantage of the adaptive precision block-Jacobi over a standard block-Jacobi using double precision. For this purpose, we specifically focus on the energy efficiency, as this has been identified as an important metric (on par with performance) for future exascale systems.

In terms of energy consumption, the accesses to main memory (MEMOPS) are at least an order of magnitude more expensive than FLOPs, and this gap is expected to increase in future systems [12]. For this reason, in the energy model, we ignore the arithmetic operations (including the access to the data in the processor registers as well as caches) and consider the data transfers from main memory only. Our energy model for estimating the global energy consumption of the solver builds on the premise that the energy cost of memory accesses is linearly dependent on the bit length of the data.

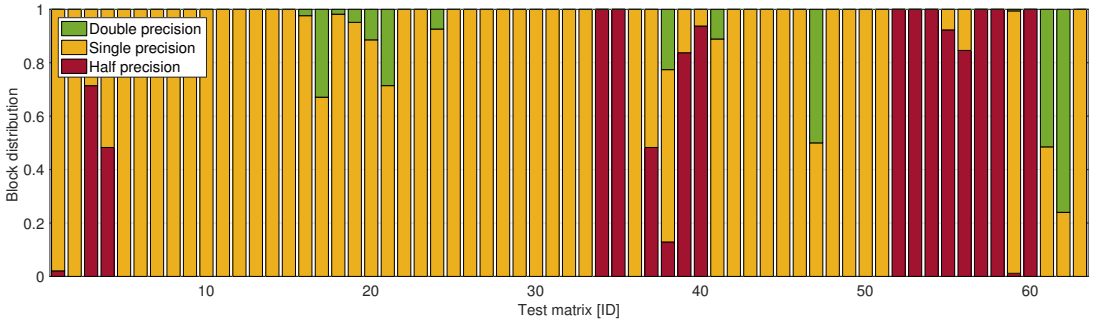

Figure 7. Details on the adaptive precision block-Jacobi. Breakdown of the diagonal blocks stored in fp64, fp32, or fp16.

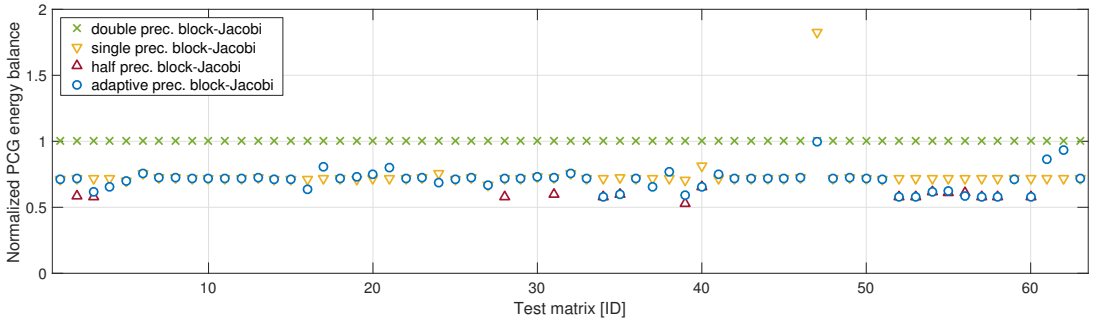

Figure 8. Energy efficiency analysis of the PCG with block-Jacobi preconditioning using different floating point formats for storing the preconditioner. The energy cost of all methods is normalized to the energy cost of the standard implementation using fp64 for storing the block-Jacobi preconditioner.

Furthermore, as we only aim to estimate the energy efficiency of the adaptive precision block–Jacobi preconditioner *relative* to the standard fp64 block-Jacobi preconditioner, we will set the (normalized) energy cost of accessing a single bit of data as 1 (energy-unit). The precision formats we consider employ 64, 32, and 16 bits.

For a problem of size n with n_z nonzero elements, the PCG method presented in Section 2 and preconditioned with a block-Jacobi preconditioner (consisting of N diagonal blocks of dimensions $m_1 \times m_1, \ldots, m_N \times m_n$) performs:

$$\underbrace{14n \cdot \text{fp64}}_{\text{vector memory transfers}} + \underbrace{(2n + n_z) \cdot \text{fp64} + (n + n_z) \cdot \text{int32}}_{\text{CSR}-\text{SpMV memory transfers}} + \underbrace{2n \cdot \text{fp64} + \sum_{i=1}^{N} m_i^2 \cdot \text{fpxx}_i}_{\text{preconditioner memory transfers}} \quad (11)$$

data transfers (from memory) per iteration, where fpxx_i denotes the precision format selected for the i-th diagonal block of the preconditioner. The data transfer volume of the block-Jacobi preconditioner thus depends on the format employed to store the block inverse. For example, with the PCG running in fp64, the approach also employs fp64 to maintain the block-Jacobi preconditioner. Further, we also consider variants that store the preconditioner entirely in fp32 or fp16 and a more sophisticated strategy that adapts the format of the distinct preconditioner blocks to the data.

For the adaptive precision block–Jacobi approach, we visualize the use of fp64, fp32, and fp16 for storing the diagonal blocks (Figure 7). Comparing this information with the data in Figure 6, we can identify a relationship between the conditioning of the blocks and the storage precision format: fp64 is primarily employed for those cases comprising very ill-conditioned blocks. Furthermore, the information in Figure 7 also shows the savings that can be attained in terms of (1) memory usage

to store the preconditioner and (2) data transfers per iteration to retrieve data from main memory. However, note that these savings do not take into account the total cost of the PCG method but only those costs strictly associated with the preconditioner application. Furthermore, the data in Figure 7 does not reflect the potentially slower convergence caused by using reduced-precision storage.

To avoid the previous two pitfalls, in our final experiment we compute the total data transfers of a single iteration of the PCG method with the block-Jacobi preconditioner stored in fp64, fp32, fp16, or adaptive precision, see equation (11). To obtain an estimated total data transfer volume, we then combine the data transfer volume per iteration with the number of iterations needed to reach convergence in each case—ignoring those cases for which half precision does not converge.

In Figure 8, we show the total energy balance *relative* to the standard approach that maintains the block-Jacobi preconditioner in double precision.

Some key observations from this last experiment are listed below.

- Storing the block-Jacobi preconditioner in fp32 often reduces the total energy cost. However, for those cases where the information loss increases the PCG iteration count, storing the preconditioner in fp32 can have a negative impact on the energy balance.
- For the (few) cases where the block-inverse matrix can be stored in fp16 without the PCG losing convergence, the total energy cost can be decreased by up to 50%.
- Using the adaptive precision block–Jacobi preconditioner never increases the total energy consumption.
- In most cases, the adaptive precision block–Jacobi preconditioner matches or outperforms the efficiency of storing the preconditioner in fp32. If the problem characteristics allow for it, the adaptive precision block–Jacobi preconditioner employs fp16 to match the half-precision efficiency while maintaining convergence for the other cases.
- The adaptive precision block–Jacobi preconditioner "automatically" detects the need to store a diagonal block in fp64 to avoid convergence degradation.

Finally, we note that, for memory-bounded operations like the block-Jacobi preconditioned CG considered here, the performance is largely determined by the data transfer volume. Therefore, the results shown in Figure 8 and the insights gained from that experiment carry over to the runtime performance of the adaptive precision block-Jacobi preconditioner. In summary, these experiments prove that the adaptive precision block–Jacobi preconditioner is an efficient strategy for improving the resource usage, energy consumption, and runtime performance of iterative solvers for sparse linear systems.

7. CONCLUDING REMARKS AND FUTURE WORK

We proposed and validated a strategy to reduce the data transfer volume in a block–Jacobi preconditioner. Concretely, our technique individually selects an appropriate precision format to store the distinct blocks of the preconditioner based on their characteristics but performs all arithmetic (including the generation of the preconditioner) in fp64. We note that the condition numbers can be obtained cheaply as our preconditioner is based on explicit inversion of the diagonal blocks. Furthermore, the overhead from selecting the appropriate storage format in the preconditioner setup can easily be amortized by the reduced cost of the preconditioner application in the solver iterations.

Our experimental simulation using Octave on an Intel architecture shows that, in most cases, storing a block–Jacobi preconditioner in fp32 has only a mild impact on the preconditioner quality. On the other hand, the reckless use of fp16 to store a block-Jacobi preconditioner fails in most cases and is therefore not recommended. The adaptive precision block–Jacobi preconditioner basically matches the convergence rate of the conventional double-precision preconditioner in all cases and automatically adapts the precision to be used on an individual basis. As a result, the adaptive-precision preconditioner can decide to store some of the blocks at precisions even less than fp32, thereby outperforming a fixed-precision strategy that relies on only a single precision in terms of data transfers and, consequently, energy consumption.

As part of our future work, we plan to investigate the effect of using other, non IEEE-compliant data formats in the adaptive block–Jacobi preconditioner, prioritizing the exponent range at the cost of reducing the bits dedicated to the mantissa. In this endeavor, we expect to reduce the problems with underflows/overflows while maintaining the "balancing" properties of the preconditioner. Furthermore, we will also develop a practical implementation of the adaptive precision block–Jacobi using IEEE formats with 16, 32, and 64 bits for modern GPUs.

ACKNOWLEDGEMENT

We thank Matthias Bollhöfer for fruitful discussions on flexible variants of Krylov solvers allowing for non-constant preconditioning operators and for pointing us to the flexible version of CG in [15]. H. Anzt was supported by the "Impuls und Vernetzungsfond of the Helmholtz Association" under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO and FEDER and the H2020 EU FETHPC Project 732631 "OPRECOMP".

REFERENCES

1. Saad Y. *Iterative Methods for Sparse Linear Systems*. 2nd edn., SIAM, 2003.
2. Anzt H, Dongarra J, Flegar G, Quintana-Ortí ES. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. *8th Int. Workshop Programming Models & Appl. for Multicores & Manycores*, PMAM, 2017; 1–10.
3. Anzt H, Dongarra J, Flegar G, Quintana-Ortí ES. Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning. *International Conference on Parallel Processing*, ICPP, 2017. To appear.
4. Dongarra J, *et al.*. Applied mathematics research for exascale computing. *Technical Report*, U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research Program 2014. https://science.energy.gov/~/media/ascr/pdf/research/am/docs/EMWGreport.pdf.
5. Duranton M, De Bosschere K, Cohen A, Maebe J, Munk H. HiPEAC vision 2015 2015. https://www.hipeac.org/publications/vision/.
6. Lucas R, *et al.*. Top ten Exascale research challenges 2014. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.
7. Lavignon JF, *et al.*. ETP4HPC strategic research agenda achieving HPC leadership in Europe 2013. http://www.etp4hpc.eu/.
8. Davis TA, Hu Y. The University of Florida Sparse Matrix Collection. *ACM Trans. on Mathematical Software* 2011; **38**(1):1–25, doi:10.1145/2049662.2049663.
9. Arioli M, Duff I. Using FGMRES to obtain backward stability in mixed precision. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]* 2008; **33**:31–44. URL http://eudml.org/doc/130614.
10. Carson E, Higham NJ. A New Analysis of Iterative Refinement and its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems. *2017.12*, MIMS EPrint, University of Manchester 2017. URL http://eprints.ma.man.ac.uk/2537/01/covered/MIMS_ep2017_12.pdf.
11. Carson E, Higham NJ. Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. *Technical Report 2017.24*, MIMS EPrint, University of Manchester 2017. URL http://eprints.ma.man.ac.uk/2562/01/paper.pdf.
12. Shalf J. The evolution of programming models in response to energy efficiency constraints. http://www.oscer.ou.edu/Symposium2013/oksupercompsymp2013_talk_shalf_20131002.pdf October 2013. Slides presented at Oklahoma Supercomputing Symposium 2013.
13. Golub GH, Ye Q. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM Journal on Scientific Computing* 1999; **21**(4):1305–1320, doi:10.1137/S1064827597323415. URL https://doi.org/10.1137/S1064827597323415.
14. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, der Vorst HV. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM: Philadelphia, PA, 1994.
15. Notay Y. Flexible Conjugate Gradients. *SIAM J. Scientific Computing* 2000; **22**(4):1444–1460.
16. Knyazev AV, Lashuk I. Steepest descent and Conjugate Gradient methods with variable preconditioning. *SIAM Journal on Matrix Analysis and Applications* 2008; **29**(4):1267–1280, doi:10.1137/060675290. URL https://doi.org/10.1137/060675290.
17. Golub GH, Van Loan CF. *Matrix Computations*. 3rd edn., The Johns Hopkins University Press: Baltimore, 1996.
18. Du Croz JJ, Higham NJ. Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis* 1992; **12**(1):1–19, doi:10.1093/imanum/12.1.1. URL http://dx.doi.org/10.1093/imanum/12.1.1.
19. Higham NJ. *Accuracy and Stability of Numerical Algorithms*. Second edn., Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002.
20. Chow E, Scott J. On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Technical Report Technical Report RAL-P-2016-006*, Rutherford Appleton Laboratory 2016.