Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software

GORAN FLEGAR, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Spain
HARTWIG ANZT, Karlsruhe Institute of Technology, Germany and University of Tennessee, USA
TERRY COJEAN, Karlsruhe Institute of Technology, Germany
ENRIQUE S. QUINTANA-ORTÍ, Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Spain

The use of mixed precision in numerical algorithms is a promising strategy for accelerating scientific applications. In particular, the adoption of specialized hardware and data formats for low precision arithmetic in high-end GPUs (graphics processing units) has motivated numerous efforts aiming at carefully reducing the working precision in order to speed up the computations. For algorithms whose performance is bound by the memory bandwidth, the idea of compressing its data before (and after) memory accesses has received considerable attention. One idea is to store an approximate operator –like a preconditioner– in lower than working precision hopefully without impacting the algorithm output. We realize the first high performance implementation of an adaptive precision block-Jacobi preconditioner which selects the precision format used to store the preconditioner data on-the-fly, taking into account the numerical properties of the individual preconditioner blocks. We implement the adaptive block-Jacobi preconditioner as production-ready functionality in the Ginkgo linear algebra library, considering not only the precision formats that are part of the IEEE standard, but also customized formats which optimize the length of exponent and significand to the characteristics of the preconditioner blocks. Experiments run on a state-of-the-art GPU accelerator show that our implementation offers attractive runtime savings.

CCS Concepts: • **Mathematics of computing → Mathematical software**; *Arbitrary-precision arithmetic*;

Additional Key Words and Phrases: Sparse linear algebra, adaptive precision, preconditioning, block-Jacobi, Krylov solvers, GPU

ACM Reference Format:
Goran Flegar, Hartwig Anzt, Terry Cojean, and Enrique S. Quintana-Ortí. 2020. Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software. *ACM Trans. Math. Softw.* 1, 1 (August 2020), 27 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Improving the robustness and speed of iterative sparse linear system solvers has been an important research topic for more than a decade. As a result, Krylov subspace methods (KSMs) are nowadays among the most efficient algorithms for large and sparse linear systems. When applied to a linear

Authors' addresses: Goran Flegar, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Castellón, Spain, flegar@uji.es; Hartwig Anzt, Karlsruhe Institute of Technology, Karlsruhe, Germany, University of Tennessee, Knoxville (TN), USA, hartwig.anzt@kit.edu; Terry Cojean, Karlsruhe Institute of Technology, Karlsruhe, Germany, terry.cojean@kit.edu; Enrique S. Quintana-Ortí, Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Valencia, Spain, quintana@disca.upv.es.

0098-3500/2020/8-ART $15.00
https://doi.org/0000001.0000001

system $Ax = b$ (with sparse coefficient matrix $A \in \mathbb{R}^{n \times n}$, right-hand side $b \in \mathbb{R}^n$, and unknown $x \in \mathbb{R}^n$) KSMs started with an initial guess x_0 produce a sequence of vectors $x_1, x_2, x_3, \ldots \in \mathbb{R}^n$ that, in general, progressively reduce the norm of the residuals $r_k = b - Ax_k$, eventually yielding an acceptable approximation to the solution of the system.

The optimization of KSMs with respect to numerical robustness and runtime performance can proceed hand-in-hand, for example, with the use of a sophisticated preconditioner. The motivation behind is that the convergence of KSMs is largely dictated by the condition number of the system coefficient matrix A. Preconditioning schemes aim to accelerate the convergence of this type of solvers by transforming the original problem into the alternative preconditioned system $(M^{-1}A)x = M^{-1}b$. An ideal preconditioner $M^{-1} \in \mathbb{R}^{n \times n}$ yields a transformed coefficient matrix $\hat{A} = M^{-1}A$ with a lower condition number than A, while admitting a software realization of the preconditioner calculation that is relatively cheap to compute and inexpensive to apply.

An example of a preconditioner typically improving both robustness and speed is the Jacobi preconditioner, and its straight-forward extension to a block-Jacobi preconditioner [Saad 2003]. The underlying inversion of the (block-)diagonal of the system matrix exhibits a high degree of parallelism while offering superior convergence acceleration when applied to problems that exhibit some inherent block structure. For example, this is the case for problems arising from a finite element discretization of a partial differential equation (PDE) [Anzt et al. 2017a].

Other optimization strategies aim at improving only runtime performance, potentially even allowing for some loss in the numerical robustness. One strategy that recently gained significant attention takes advantage of lower precision formats in parts of the algorithm [Abdelfattah et al. 2020]. The motivation for this idea is that KSMs, enhanced with some form of a simple preconditioner, are memory-bound algorithms, implying that their performance on current architectures is constrained by the bandwidth between the floating-point units (FPUs) and the memory where the data resides. In case the problem data is too large to fit into the cache memory of the processor(s), the increasing gap between the throughputs of the processor and the main memory (also known as the *memory wall* [Dongarra et al. 2014; Lucas et al. 2014],) dictates the performance of this type of algorithms. This is a well-recognized problem, especially in the domain of sparse linear algebra operations, where *communication-avoiding* techniques are particularly appealing; see, e.g., [Cools 2018; Hoemmen 2010] and the references therein.

The idea of mixed precision KSMs tackles the memory bottleneck by reducing the communication volume and memory footprint. For example, the authors of [Carson and Higham 2018] diminish data movement (and arithmetic cost) using the standard IEEE half/single/double precision formats [Commitee 2000] in combination with iterative refinement.[1] Other efforts aim at reducing the indexing information necessary to maintain the sparse system matrix, e.g. via the compressed storage block (CSB) format [Buluç et al. 2009].

A technique orthogonal to these efforts targets not the KSM, but the preconditioner itself. In [Anzt et al. 2019], we proposed to reduce the pressure on the memory bandwidth by adjusting the precision format used to store the preconditioner [Anzt et al. 2019]. We analyzed the approach under theoretical aspects for a CG solver equipped with a block-Jacobi preconditioner that operates (that is, performs all arithmetic) in full double precision, while accessing the inverted diagonal blocks of the block-Jacobi preconditioner in a problem-adapted (potentially lower) precision. More precisely, all the problem data is stored in IEEE double precision format, except the blocks of the preconditioner, which are stored in either IEEE half/single/double precision formats, depending on their condition numbers. A type transformation is therefore required every time the preconditioner

[1] In the setting of the solution of linear systems, iterative refinement is an old technique, which dates back to the use of the first desk calculators, in the 1940s [Higham 2002].

blocks stored in half or single precision in main memory are moved to the registers (where they are maintained in double precision). The theoretical data transfer savings were estimated using an analytical model that takes the floating point format and the convergence impact into account. For a significant portion of the symmetric positive definite matrices available in the SuiteSparse Matrix Collection [Davis and Hu 2011], we observed data transfer savings of up to 70% compared with a solver that handles all (preconditioner) data and arithmetic using double-precision.

In this paper, we build upon our preliminary theoretical analysis by deriving the first practical implementation of the adaptive precision block-Jacobi preconditioner, proving the practical usability in the context of high performance computing on state-of-the-art GPU architectures, and disseminating the production-ready implementation along with usage examples in the Ginkgo numerical linear algebra library.

Specifically, we make the following specific contributions:

(1) We move from a theoretical analysis of the usability and potential performance benefits [Anzt et al. 2019] to an actual implementation of the adaptive precision block-Jacobi preconditioner, ready to run on high-end GPUs, which leverages an ample variety of hardware-specific optimization techniques ranging from cache-line alignment to cooperative group communication.
(2) We extend the idea presented in [Anzt et al. 2019] by adopting also precision formats outside the IEEE standard to optimize the length of exponent and significand to the problem properties.
(3) We derive algorithm-specific kernels that entail the extraction of the diagonal blocks, the inversion of the diagonal blocks via Gauss-Jordan elimination featuring pivoting [Anzt et al. 2018], the computation of the condition number and the data range [Anzt et al. 2018], and the selection of the optimal storage precision. These kernels are needed for the adaptive precision block-Jacobi preconditioner generation, and they are heavily optimized to incur only negligible overhead compared to a standard block-Jacobi preconditioner generation.
(4) We propose an efficient compact layout to store the blocks of the adaptive precision block-Jacobi preconditioner that optimized the memory access.
(5) We evaluate the performance of a production code realizing the adaptive precision block-Jacobi preconditioner scheme in the framework of a high-performance CG implementation on a NVIDIA Volta GPU. This experimental evaluation demonstrates the validity of the approach and reveals up to 30% performance improvement over a standard (double precision) block-Jacobi preconditioner for a large range of real-world test problems.
(6) We deploy the production-ready block-Jacobi preconditioner in the Ginkgo numerical linear algebra library along with usage examples.

Our approach shares some of the appealing properties of the prototype in [Anzt et al. 2019]. Concretely, we employ full double precision in the generation and application of the preconditioner, as well as in all other arithmetic computations. Furthermore, we store part of the preconditioner in reduced precision, and convert it into full precision before proceeding with the arithmetic operations in the actual preconditioner application. Thus, our preconditioner still ensures that the preconditioning operator preserves orthogonality in double precision, implying that previously orthogonal Krylov vectors are orthogonal after the preconditioner application. In consequence, there is no need for flexible variants that introduce an additional orthogonalization step to preserve convergence [Golub and Ye 1999].

The rest of the paper is structured as follows. In Section 2.2 we introduce the Ginkgo numerical linear algebra library and briefly review the idea of KSMs and block-Jacobi preconditioning. More details about the idea of decoupling the memory precision from the arithmetic precision [Anzt et al. 2019] and the adaptive precision block-Jacobi are presented in Section 3. We elaborate on the

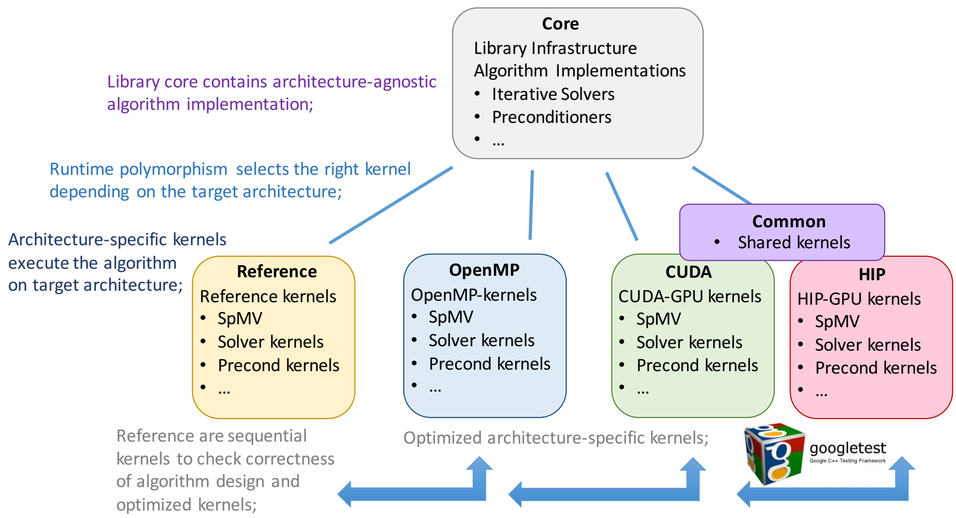

Fig. 1. The Ginkgo library design overview with the library core separated from the architecture-specific backends for AMD GPUs (hip), NVIDIA GPUs (cuda), multicore (omp), and the reference backend for correctness checks [Anzt et al. 2020b].

first high performance realization of the adaptive precision block-Jacobi in Section 4. We dedicate Section 5 to motivate the need for making novel algorithms and high performance implementations available in sustainable open source software. In Section 6, we present performance results for the block-Jacobi preconditioner generation and application, and analyze the effectiveness and efficiency of the adaptive precision block-Jacobi preconditioner. Next, In Section 7 we discuss some central aspects of adaptive precision preconditioning in general and the experimental results in particular, and conclude in Section 8 with a summary of the findings and future research directions.

2 HIGH PERFORMANCE SPARSE LINEAR ALGEBRA ON GPUS

2.1 The Ginkgo numerical linear algebra library

Ginkgo [Anzt et al. 2020a] is a modern sparse linear algebra library implemented in C++ that embraces two principal design concepts: The first principle, aiming at future technology readiness, is to consequently separate the numerical algorithms from the hardware-specific kernel implementation to ensure correctness (via comparison with sequential reference kernels), performance portability (by applying hardware-specific kernel optimizations), and extensibility (via kernel backends for other hardware architectures). The second design principle –aiming at user-friendliness– is the convention to express functionality in terms of linear operators: every solver, preconditioner, factorization, matrix-vector product, and matrix reordering is expressed as a linear operator (or composition thereof).

A high-level overview of Ginkgo's software architecture is displayed in Figure 1 [Anzt et al. 2020b]. The library design collects all classes and generic algorithm skeletons in the "core" library which are accessed via the driver kernels available in th "cuda," "hip," "omp," and "reference" modules. We note that "reference" contains sequential CPU kernels used to validate the correctness of the algorithms and serve as a reference implementation for the unit tests realized using the googletest framework [Google Google]. The "cuda," "hip," and "omp" modules are heavily optimized kernel backends for NVIDIA GPUs, AMD GPUs, and multicore CPUs, respectively.

GINKGO relies on the "executor" concept to enable platform portability. The executor specifies the memory location and the execution space of the linear algebra objects and represents computational capabilities of distinct devices. Each executor implements methods for allocating/deallocating memory on the device targeted by that executor, copying data between executors, providing hardware-specific kernels, running operations, and synchronizing all operations launched on the executor. As all information in the executor is encapsulated and all memory allocation and kernel selection is automatically orchestrated, the user can run a single code on different platforms without having to modify the code by selecting a different executor in the beginning of the application.

2.2 Computational Aspects of KSMs and block-Jacobi Preconditioning

Most instances of KSMs, such as CG, BiCG, GMRES, BiCGStab, etc., are comprised of a sequence of calls to simple computational kernels, such as the dot or inner product (DOT), AXPY-like vector updates, and the sparse matrix-vector product (SPMV), inside an iteration loop [Saad 2003]. These kernels are all memory-bound operations, with a ratio between floating-point operations (FLOPs) and memory accesses (MEMOPs) that is $O(1)$, globally yielding a memory-bound solver.

Block-Jacobi preconditioners split the coefficient matrix into $A = L + M + U$, where the preconditioner defined by $M = \text{diag}(D_1, D_2, \ldots, D_m) \in \mathbb{R}^{n \times n}$, with $D_i \in \mathbb{R}^{m_i \times m_i}$ and $\sum_{i=1}^{m} m_i = n$, is a block-diagonal matrix containing the corresponding entries on the diagonal blocks of A, while $L, U \in \mathbb{R}^{n \times n}$ contain the elements of the coefficient matrix below and above those of M, respectively. (The scalar Jacobi preconditioner is a simple variant of the block counterparts with $m_i = 1$, $i = 1, 2, \ldots, m$, so that M only contains the diagonal of A.) The block-Jacobi preconditioner is well defined if the diagonal blocks D_i are all nonsingular. Furthermore, block-Jacobi preconditioning is particularly effective if the system matrix A inherently presents a block structure (which is the case for many problems that arise from a finite element discretization of a PDE [Anzt et al. 2017a]) that is matched by the block structure of the Jacobi preconditioner.

In this work, we integrate a block-Jacobi preconditioner that explicitly computes the block-inverse matrix, $M^{-1} = \text{diag}(D_1^{-1}, D_2^{-1}, \ldots, D_m^{-1}) = \text{diag}(E_1, E_2, \ldots, E_m)$, before the iteration process of the KSM commences. The preconditioner is then applied within the KSM iteration in terms of a dense matrix-vector multiplication (GEMV) per inverse block E_i. Thus, the iteration for the preconditioned KSM remains a memory-bound process, as so is the GEMV kernel, independently of the block size m_i. In practice, the resulting preconditioner is of a comparable quality to the one computed by the conventional (and numerically more stable) strategy that computes the LU factorization (with partial pivoting) [Golub and Van Loan 1996] of each block ($D_i = L_i U_i$), and then applies the preconditioner using two triangular solves (per factorized block)[Anzt et al. 2018, 2017]. In exchange for a higher cost, the block-Jacobi preconditioner with explicit computation of the inverses presents the appealing property of yielding an application based on a highly parallel kernel (GEMV), compared with the constrained parallelism of the triangular systems that are necessary in the application of the LU-based preconditioning counterpart [Anzt et al. 2017b].

3 ADAPTIVE BLOCK-JACOBI PRECONDITIONING

3.1 Standard IEEE precision formats

In [Anzt et al. 2019], we proposed an adaptive block-Jacobi preconditioner that individually tunes the storage format of each block D_i depending on its condition number. The scheme adopted in that work relies on three precision formats: 16-bit (fp16), 32-bit (fp32) and 64-bit (fp64), which correspond to the standard IEEE half, single and double precision formats [Commitee 2000], respectively. In detail, the adaptive block-Jacobi preconditioner proceeds as follows:

(1) Before the iteration commences, we explicitly compute the inverse of each block using fp64: $D_i \longrightarrow E_i$.
(2) At the same stage (i.e., before the iterative solver is started), we compute $\kappa_1(D_i) = \kappa_1(E_i) = \|D_i\|_1 \|D_i^{-1}\|_1 = \|D_i\|_1 \|E_i\|_1$. As E_i is explicitly available, computing $\kappa_1(D_i)$ is straightforward and inexpensive compared with the inversion of the block [Anzt et al. 2018].
(3) After inverting the diagonal block D_i in fp64, we store the inverted diagonal block E_i in the format determined by its condition number—truncating the entries of the block if necessary. Precisely, we store E_i in

$$\begin{cases} \text{fp16} & \text{if } \tau_h^L < \kappa_1(D_i) \le \tau_h^U, \\ \text{fp32} & \text{if } \tau_s^L < \kappa_1(D_i) \le \tau_s^U, \text{ and} \\ \text{fp64} & \text{otherwise,} \end{cases} \tag{1}$$

where the thresholds τ are set as $\tau_h^L = 0$ and $\tau_h^U = \tau_s^L$.
(4) During the iteration, we recover the block E_i, stored in the corresponding format in memory (as determined by (1)), transform its entries to fp64 in the processor's registers, and apply the block in terms of a fp64 GeMV.

A central aspect is the choice of the values for τ, which is strongly related to the question of how much accuracy of the preconditioner should be preserved. For preserving the accuracy a of the preconditioner (e.g., $a = 10^{-1}$), a storage format with round-off error u can be considered valid for a block D if $\kappa(D) \le a/u$. Furthermore, the value τ for this format is computed as $\tau = a/u$. While the round-off errors u are format-specific, but fixed, the values of τ are still variable with respect to how much accuracy of the preconditioner should be preserved.

Due to the use of the standard formats for half, single and double precision, in the above procedure the truncation can result in either overflows or underflows, whose consequences need to be tackled. Here we only discuss the second case and refer the reader to [Anzt et al. 2019] for the handling of overflows. The risk associated with underflow is that the truncation may turn a non-zero (but close to zero) value in fp64 into a zero which in turn can make E_i an ill-conditioned (or even singular) block, thereby causing numerical difficulties for the convergence of the KSM. In order to avoid this issue, we examine the condition number of the truncated representation of E_i, and discard the use of the corresponding reduced precision if it was above a given threshold τ_κ.

3.2 Unconventional precision formats

In addition to the three floating point formats defined by the IEEE standard, this work augments the set of considered precisions with three additional formats that can be cheaply processed using the instruction set of NVIDIA GPUs. While it would be theoretically possible to employ any combination of exponent and significand bits, the complexity of purely software-based format conversion could prove detrimental to performance. However, conversions for several particular precision configurations can be implemented efficiently.

In particular, if the conversion to lower precision preserves the number of exponent bits and the rounding mode is limited to *round-to-zero*, the conversion to lower precision consists of significand truncation, only. Converting back to full precision then conversely adds zeros as the missing significand bits[Anzt et al. 2019]. Using the notation $fp_{ex,snf}$ (where *ex* and *snf* denotes the number of exponent bits and significand bits, respectively), this procedure can be used on the 64 bit IEEE double precision format ($fp_{11,52}$) with 11 exponent and 52 significand bits to obtain an alternative 32 bit floating point format with 11 exponent and 20 significand bits ($fp_{11,20}$) by dropping the 32 low-order bits of the original format. The range of such a format stays roughly the same as that of IEEE double precision, and the unit roundoff (adjusted for the *round-to-zero* rounding mode) is

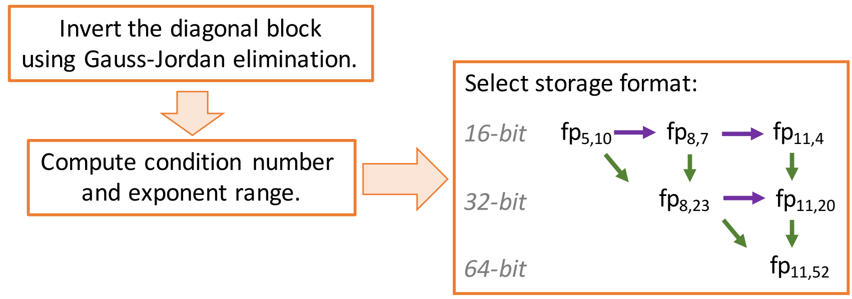

Fig. 2. Workflow for generating the inverse block and selecting a suitable storage format. The horizontal arrows (purple) reflect bitcount-constant traversals addressing overflow and underflow, the green vertical arrows represent significand extensions for increasing the accuracy to the requirements imposed by the condition number.

$u = 9.54e-7$. A 16-bit format based on IEEE double ($fp_{11,4}$) can also be obtained by dropping the 48 low-order significand bits. The result is a format with 11 exponent and 4 significand bits and unit roundoff $u = 6.25e-2$. A 16-bit format can also be obtained by basing it on the 32 bit IEEE single precision ($fp_{8,23}$). Such a format ($fp_{8,7}$) has 8 exponent and 7 significand bits, and unit roundoff of $u = 7.81e-3$.

The additional formats offer a trade-off by providing formats of the same size as their IEEE counterparts, but with larger range and lower precision. They can be used to store a block that is relatively well conditioned (and thus does not require high precision to achieve reasonable accuracy [Anzt et al. 2019]), but the range of values in the block is such that a conventional format would cause catastrophic overflows or underflows. The improved format selection strategy selects the first format from the list $fp_{5,10}$, $fp_{8,7}$, $fp_{11,4}$, $fp_{8,23}$, $fp_{11,20}$, $fp_{11,52}$ whose unit roundoff is small enough to deliver the required accuracy, and where the exponent range avoids catastrophic overflows and underflows. The list is sorted by increasing sizes of the formats, which means that the procedure selects the smallest format capable of delivering the required accuracy. Within the same format size, the list is sorted so that priority is given to the format that offers more accuracy. In Figure 2 we visualize the process of generating the block-Jacobi preconditioner and selecting a suitable storage format.

4 CUDA IMPLEMENTATION

4.1 Previous work

As a starting point for the implementation of adaptive block-Jacobi kernels, we use a previous prototype CUDA implementation of full precision block-Jacobi [Anzt et al. 2018]. The implementation includes an optimized kernel for block-Jacobi preconditioner generation which extracts the diagonal blocks from the sparse system matrix stored in Compressed Sparse Row (CSR [Saad 2003]) format, inverts them, and stores the inverses into the GPU main memory. For each block, the entire pipeline is executed using a single warp (a group of 32 GPU cores, roughly equivalent to a 32-wide vector unit) with each core processing a single column of the matrix. The inversion is realized via the highly parallel Gauss-Jordan Elimination (GJE) algorithm, and the explicit inverse diagonal blocks are stored in row-major order to enable coalesced access both when extracting the blocks from the sparse structure, as well as when storing the inverses back into memory. The generation pipeline leverages the extensive register storage available in recent CUDA architectures (up to 32 KB per warp) to keep the entire block in processor registers during the computation and completely

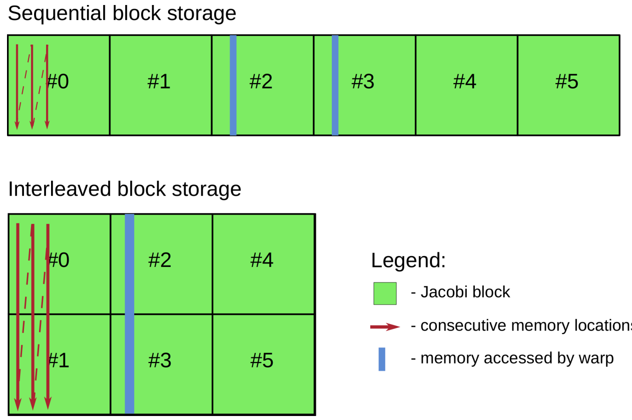

Fig. 3. Preconditioner storage scheme. Top: sequential storage used by the initial implementation. Bottom: block-interleaved storage used by the new implementation.

avoid expensive data access. This strategy allows to efficiently process double precision blocks of up to 32 rows and columns.

The second component of the prototype is a custom implementation of the preconditioner application procedure. Once again, each warp is responsible for processing a single preconditioner block. First, the section of the input vector corresponding to the block is read into the registers and distributed among the threads of the warp. Then, for each row of the block, the warp collaboratively reads the values in the row, forms a dot product between the input vector (already present in the registers) and the row, and writes the result to the output vector. Processing the blocks stored in row-major in this way ensures contiguous access to the main memory.

A final optimization included in the initial prototype involves the processing of small blocks. If all the preconditioner blocks are smaller than some dimension $k < 32$, a more efficient version of the kernel can be generated by having each thread of the warp use an array large enough to store only k instead of 32 values. This reduces the resource requirements of the warp, allowing the GPU to simultaneously process more warps per multiprocessor. In addition, for small values of k, a warp can be logically split into two (or more) sub-warps; then, instead of using the entire warp to process a single block, each sub-warp can handle the generation of one preconditioner block. Precisely, for a maximum block size $\hat{k}$, every warp handles $2^{5-\lceil \log_2 \hat{k} \rceil}$ blocks:

$$
\begin{array}{ll}
32 \geq \hat{k} > 16 & \text{1 block per warp,} \\
16 \geq \hat{k} > 8 & \text{2 blocks per warp,} \\
8 \geq \hat{k} > 4 & \text{4 blocks per warp,} \\
4 \geq \hat{k} > 2 & \text{8 blocks per warp,} \\
2 \geq \hat{k} > 1 & \text{16 blocks per warp,} \\
1 = s & \text{32 blocks per warp.}
\end{array}
$$

To enable these optimizations, we generate a kernel optimized for each maximal block size $\hat{k} = 1, 2, \ldots, 32$.

4.2 Kernel improvements

Before implementing the adaptive precision version of the block-Jacobi preconditioner, we first incorporate several improvements to the full precision block-Jacobi preconditioner.

Starting with CUDA toolkit version 9.0, NVIDIA updated the warp shuffle and warp vote APIs used for intra-warp communication to support the new Volta architecture that features relaxed warp execution constraints [NVIDIA Corporation 2018]. While the APIs used by the previous implementation of block-Jacobi kernels are still available (albeit deprecated), using them causes the kernel to stall[2] when run on the Volta architecture. In addition to the updated low-level APIs, the CUDA toolkit version 9.0 also includes a new cooperative group APIs which encapsulates the details of the low-level APIs. Instead of using the low level API directly, we decided to modify our code to use this high-level alternative as it provides more flexibility and can potentially enable better compatibility with future CUDA versions.

We also identified several additional performance optimizations concerning the memory layout of the block-Jacobi preconditioner, specifically the question of storing the blocks in row-major vs. column-major layout. A detailed analysis of the preconditioner application kernel explained in Section 4.1 revealed that the time needed for intra-warp communication in the collaborative computation of the dot product (necessary in a row-major block storage) is significant compared with the time needed to load the data from memory, so improving that part of the kernel can render performance gains. For this reason, we change the data layout of the preconditioner blocks to use column-major instead of row-major storage. This enables efficient column-wise access of the block – equivalent to a column-major GEMV for each Jacobi block. The downside of this approach is that the block data has to be transposed after the inversion, which results in suboptimal memory accesses during the preconditioner generation step. However, since the preconditioner is generated only once, but applied multiple times (at least once per KSM iteration), we expect this change in storage layout will render performance improvements for most use cases.

The final improvement aims at processing small blocks more efficiently. The original implementation stores consecutive blocks in sequence, as depicted in the top part of Figure 3. With such storage, memory access during preconditioner application is optimal for large blocks. However, as soon as the maximal block size becomes small enough to split the warp into sub-warps, so that several blocks are processed by the same warp, this no longer holds. Since the corresponding columns of consecutive blocks are not consecutive in memory, reading them causes suboptimal strided memory access. To eliminate this problem, we replace the sequential storage scheme with the block-interleaved storage shown in the bottom part of Figure 3. The new scheme groups all blocks processed by a warp together, and interleaves the storage of their columns. Precisely, the scheme initially stores the first columns of all blocks in the group, then proceeds with storing the second columns, etc.; this strategy ensures contiguous memory accesses during preconditioner application.

The last two optimizations are essential to enable performance improvements via low precision storage. Without the former, communication would dominate the cost of preconditioner application, severely limiting the benefit of reduced data transfers. Without the latter, accessing small blocks would incur unnecessary data loads into cache. Since the size of the cache lines is fixed, reducing the size of the individual elements would just increase the amount of memory being wasted, without reducing the total data movement volume.

---

[2]Since only a subset of the warp was calling the API in the original implementation.

4.3 Adaptive block-Jacobi

Extending the full precision block-Jacobi to the adaptive precision variant requires adding the precision detection logic to the preconditioner generation, storing the blocks in appropriate precision together with metadata specifying which precision is employed for the distinct blocks and, during preconditioner application, restoring the original block on the fly from low precision storage using the metadata.

The precision selection method we employ is that explained in Section 3.1, enhanced with the additional formats introduced in Section 3.2. The condition number of the block is determined by computing the matrix 1-norm of the block before and after inverting it [Anzt et al. 2018]. The condition number is then evaluated against the unit roundoffs to select the optimal format using the format priority list we introduced in Section 3.2. For precisions that require additional protection against catastrophic underflow or overflow (IEEE singe and half), the conditioning of the inverse stored in lower precision is computed by converting each value of the inverse block to lower precision, converting it back to double precision, followed by norm calculation, inversion, and another norm calculation — all in double precision. This way, the condition number is computed with high accuracy. Before reducing the precision, a copy of the full precision inverse is backed up to main GPU memory. This allows to retrieve the full precision inverse afterwards (if necessary). When a group of blocks is processed by a single warp (in case of small blocks), the precision is not decided individually, but jointly for the entire group of blocks, using the first precision in the list from Section 3.2, which is suitable for storing all blocks. This is done for performance reasons, as trying to execute different instructions by threads belonging to the same warp — which would be necessary to read values stored in different precisions — would lead to thread divergence, and the serialization of these instructions, ultimately resulting in a significant slowdown.

Since the final precisions are not known before inverting the blocks, a memory workspace large enough to store all blocks in double precision is allocated before launching the preconditioner generation kernel. Once the storage precision is decided, low precision blocks are stored using only the first part of the workspace they are assigned to, while the rest of the workspace remains unused (fragmentation). While it would be possible to post-process the block storage structure to remove unused "gaps" via de-fragmentation, doing so would not reduce the total memory transfer volume during preconditioner application, since the total storage required for the group of blocks is a multiple of the cache line size in any precision, as long as the block size is at least 2. In consequence, the "gaps" will never be transferred from main memory to the cache. Thus, removing gaps is only attractive in case the total memory footprint of the preconditioner is a relevant factor. We refrain from incorporating de-fragmentation in our implementation.

A distinct memory block is used to store the information about the precisions used for the inverted blocks. The precision of each block is encoded using 8 bits, which is the smallest amount of data that can be independently stored and loaded from memory. This information is retrieved during the preconditioner application stage to determine the storage locations and precision formats of individual blocks and select the correct conversion procedure.

5 USABILITY, REPRODUCIBILITY AND SUSTAINABILITY EFFORTS

As not only modern hardware but also the software that can effectively utilize the hardware resources becomes increasingly complex, it can no longer be expected that novel algorithms or high performance implementations presented in scientific publications are adequately explained so that the readers can reproduce an implementation of equivalent quality. Furthermore, domain scientists who can potentially benefit from such work, should not be required to understand low-level optimization techniques needed to produce a high performance implementation. In consequence,

```
    // Read data
    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
    auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
    auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

    // Generate solver
    auto solver_gen =
        cg::build()
+           .with_preconditioner(
+               gko::preconditioner::Jacobi<>::build().on(exec))
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(20u).on(exec),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(1e-20)
                    .on(exec))
            .on(exec);
    auto solver = solver_gen->generate(A);

    // Solve system
    solver->apply(lend(b), lend(x));
```

```
    // Read data
    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
    auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
    auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

    // Generate solver
    auto solver_gen =
        cg::build()
+           .with_preconditioner(
+               gko::preconditioner::Jacobi<>::build()
+                   .with_storage_optimization(
+                       gko::precision_reduction::autodetect())
+                   .on(exec))
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(20u).on(exec),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(1e-20)
                    .on(exec))
            .on(exec);
    auto solver = solver_gen->generate(A);

    // Solve system
    solver->apply(lend(b), lend(x));
```

Fig. 4. Changes needed to enhance Ginkgo's `simple_solver` usage example with the full precision block-Jacobi preconditioner (top) and adaptive precision block-Jacobi preconditioner (bottom).

it is becoming increasingly important to openly publish high performance implementations and simplify their integration into other software ecosystems.

To address these issues, we integrate both the full-precision block-Jacobi preconditioner as well as the adaptive precision variant into the open source Ginkgo linear algebra package[3]. Ginkgo is a C++ library originally designed for the iterative solution of sparse linear systems. It features various matrix formats and solvers with high performance implementations for both GPU and CPU architectures, and allows for the easy integration into existing software stacks. At this point, the (adaptive) block-Jacobi preconditioner is available in "reference mode" (a single threaded straightforward CPU implementation that can be used for correctness checking and evaluating the convergence benefits of the preconditioner) as well as in "CUDA mode", with the latter featuring the high performance GPU implementation described in this work. A high performance CPU implementation based on OpenMP parallelization is planned, but not yet available.

Adding the block-Jacobi preconditioner into a larger software effort provides the benefits of reusing existing workflows: Ginkgo's low-level building blocks are utilized to simplify the implementation; its unit testing framework extended to include tests for the block-Jacobi preconditioner

[3]https://ginkgo-project.github.io

which are then automatically run using Ginkgo's continuous integration (CI) system; the benchmarks for block-Jacobi are integrated into Ginkgo's Continuous Benchmarking (CB) framework and can be run separately, or in conjunction with the rest of the benchmarks [Anzt et al. 2019]. From the user's perspective, the integration reduces the amount of software that has to be installed as well as simplifies the installation (as Ginkgo uses the well-established CMake build system) and integration of the software. If the user is already using Ginkgo as part of an application, adding the adaptive block-Jacobi preconditioner requires only a few of additional lines of code, as shown in Figure 4. If the application does not (yet) use Ginkgo, its library interoperability features can be used to wrap existing data structures into Ginkgo objects, which can then be used to construct and apply the preconditioner. Finally, as all the unit tests and benchmarks contributed in this work are distributed as part of the Ginkgo ecosystem, it should be relatively simple for the user to verify the correctness and reproduce the performance results we present in this paper, or even to evaluate the kernels' performance on a different CUDA-supporting architecture.

6 EXPERIMENTAL EVALUATION

This section evaluates the numerical properties and the effectiveness, efficiency, and performance of the developed adaptive precision algorithm and the low level kernels on recent CUDA-supporting GPUs. Initially, we assess the performance gains available from the improvements to the full precision block-Jacobi preconditioner. Then, the optimized full precision kernels are compared with the adaptive precision variant.

Two hardware setups are used in the experiments. The first one is a GPU-accelerated node of a compute cluster at the University of Jaume I (UJI). The node is composed of an 8-core Intel Xeon E5-2620 v4 CPU with 32 GB of RAM and an NVIDIA TESLA P100 (PCI-e form factor) GPU with 16 GB of HBM2 memory. The accelerator achieves a peak double precision performance of 4.7 TFlop/s and a peak memory bandwidth of 732 GB/s.

The second setup is the Summit supercomputer at the Oak Ridge National Laboratory. Our experiments use a single node containing two 22-core IBM POWER9 CPUs with 256 GB of RAM and 6 NVIDIA TESLA V100 (SXM2 form factor) GPUs with 16 GB of HBM2 memory. For our experiments, we use only a single NVIDIA V100 GPU with a peak double precision performance of 7.8 TFlop/s and a peak memory bandwidth of 900 GB/s.

6.1 Effects of using the cooperative group APIs and the newer Volta architecture

While the C++ language and its compilers are designed to enable zero-overhead abstractions, there is always a possibility that a particular abstraction is not properly translated by the compiler and does not generate sufficiently optimized code. Thus, we first evaluate the effect of replacing the low-level warp shuffle and vote APIs used in the original code with the higher-level cooperative groups API. Since the initial version using the low-level APIs does not work correctly on the new Volta generation hardware used by the Summit system, the evaluation was realized on the UJI cluster, which features the older Pascal generation P100 GPU. For the new implementation that supports the recent Volta architecture, we also include the results obtained on the Summit system and the V100 GPU to study the effects of switching to newer hardware.

The experiments were performed using synthetically-generated block-diagonal matrices with varying block sizes and a total of 50,000 equally-sized blocks per matrix. The maximal size of preconditioner blocks was set to match the block size of the matrix. The nonzero locations were filled with randomly-generated small floating point numbers uniformly distributed between -1 and 1.

The results presented in Figure 5 reveal that there is no significant performance difference when moving to the higher level API. At the same time, the version using the cooperative groups API

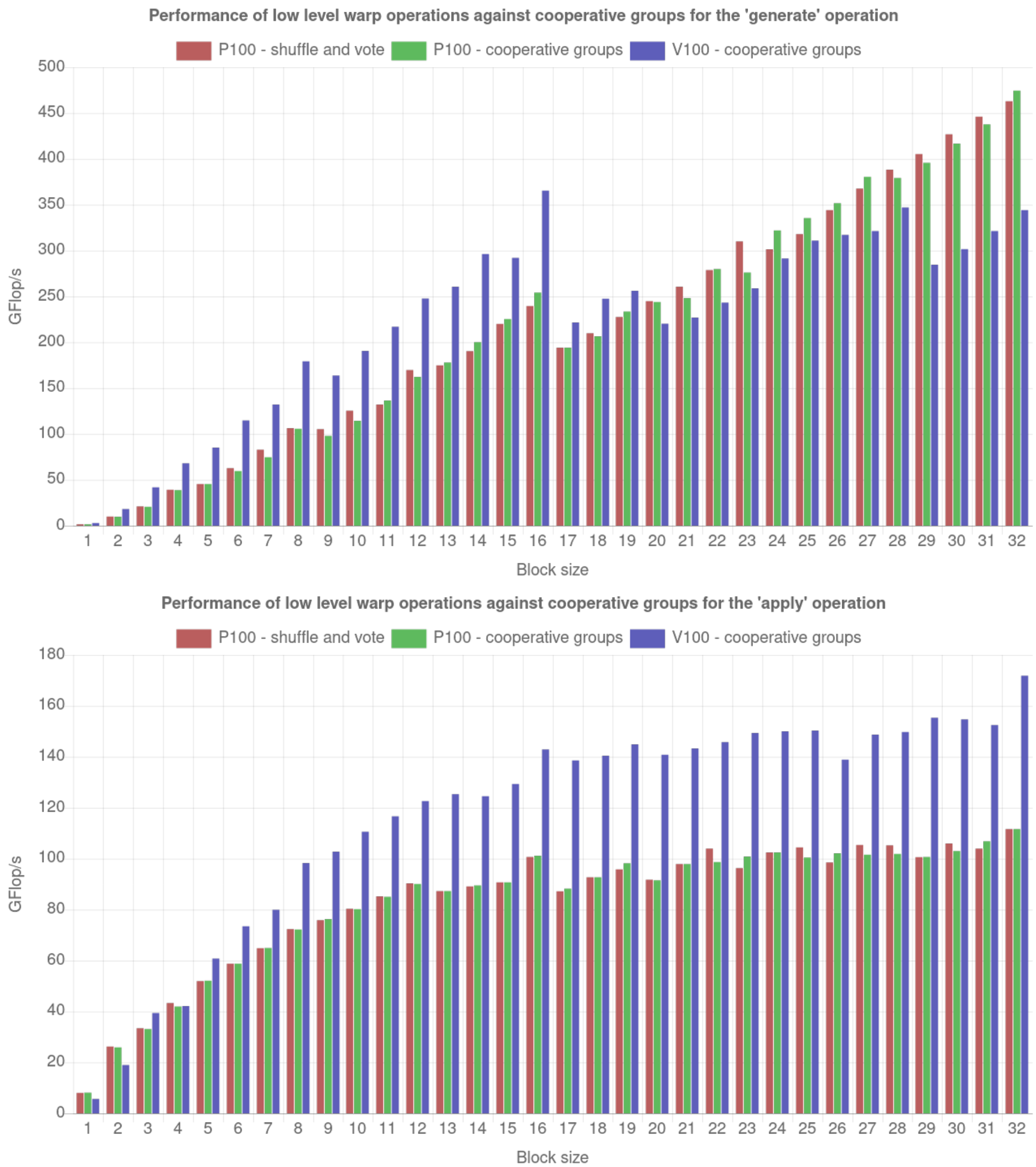

Fig. 5. Effects of using the higher-level cooperative groups API (green and blue) over the low-level warp shuffle and warp vote APIs (red). The results with cooperative groups are shown for the older P100 GPU (green) and newer V100 GPU (blue). The top plot shows the performance of the preconditioner generation (not including block size detection) and the bottom plot the performance of the preconditioner application.

supports the new Volta architecture. The preconditioner generation stage takes a slight performance hit for large blocks due to the Volta architecture increasing register pressure to support the relaxed warp execution model [Anzt et al. 2018]. However, the more relevant preconditioner application stage does not suffer from this problem, and exhibits performance improvements between 40% and 50% on the V100.

6.2 Memory layout improvements

Next, we evaluate the effect of the two preconditioner memory layout optimizations: the block-level optimization that employs column-major instead of row-major storage and the preconditioner-level optimization based on block-interleaved instead of sequential block storage. We run the experiments on Summit's V100 GPU, using the same synthetic benchmarks as in Section 6.1.

Figure 6 demonstrates that changing the storage scheme from row-major to column-major slightly reduces the performance of preconditioner generation as storing the preconditioner data causes non-coalesced memory access. On the other hand, the performance of preconditioner application increases for all block sizes, due to the availability of a more efficient matrix-vector product algorithm. As mentioned earlier, since the preconditioner is generated only once and applied multiple times (and the cost of generating the block-Jacobi preconditioner is negligible compared with the solver runtime [Anzt et al. 2018]), the overall performance of the solver is improved.

The second optimization only has impact on blocks with at most 16 rows and columns. This is expected, as for larger blocks the two storage schemes result in exactly the same data layout. For those cases where the storage layout is different, marginal benefits can be identified in favour of the block-interleaved layout. We expect that these benefits become more pronounced in the adaptive block-Jacobi variant, since unfavorable cache access is more detrimental when dealing with smaller data types.

6.3 Adaptive precision

Having analyzed the effects of the additional improvements applied to the full precision block-Jacobi, we now turn our attention to the adaptive variant. Once again, we run the experiments on Summit's V100 GPU and use synthetic benchmarks described in Section 6.1. For the full precision version, we evaluate a kernel incorporating all the optimizations described in previous sections: cooperative groups API, column-major storage and block-interleaved block storage. For the adaptive version, a kernel featuring all these optimization steps and additional support for adaptive precision is used. We report results where the autodetection system was disabled, and the precision used for all blocks is fixed beforehand to the same value. This offers an upper bound on the theoretical performance improvement that can be expected if all blocks can be stored in the same precision. On real-world problems (covered in the next section), the actual performance improvement highly depends on the condition number distribution of the diagonal blocks of the system matrix, which is difficult to replicate with synthetic benchmarks. However, since disabling autodetection means that part of the preconditioner generation kernel is skipped, we additionally report performance results that account for the autodetection: a variant that selects only between the three precision formats that do not require additional condition number calculation, and a variant with full autodetection using all six supported precisions formats.

Figure 7 shows the results for the generation and application stages of the adaptive precision block-Jacobi preconditioner and all six supported precisions. While there are some performance improvements available when using lower precision in the generation stage due to the reduction of the total time needed to store low-precision blocks, the improvements in the application stage are of higher importance. These improvements are more pronounced for larger block sizes, where the preconditioner's memory footprint becomes more relevant compared with the footprint of the input and output vectors. In total, low-precision blocks can yield up to 1.7× and 2× speedups for 32-bit and 16-bit storage schemes, respectively. Another interesting observation concerns the effect

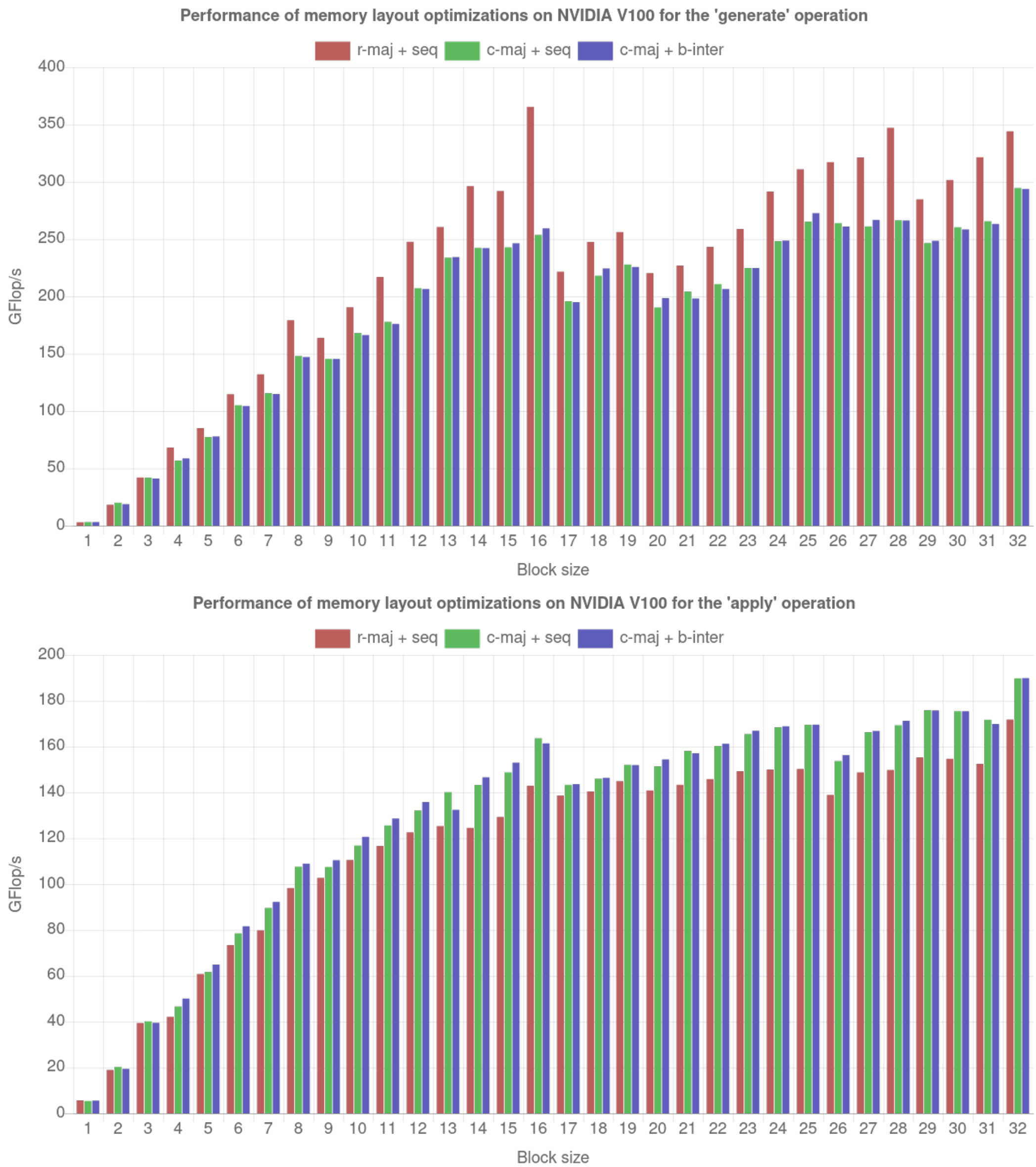

Fig. 6. Performance of memory layout optimizations on the V100 GPU. The original row-major, sequential block storage is shown in red, column-major, sequential block storage in green and column-major, block-interleaved block storage in blue. The top plot shows the performance of the preconditioner generation (without block size detection) and the bottom plot the performance of the preconditioner application.

of automatic precision detection on the preconditioner generation time. When using only the three non-standard formats that preserve the representable range of values, there is virtually no impact on the performance of preconditioner generation. On the other hand, using all six formats can lead to performance degradation of up to a factor of 2×. This implies that, in case the solver is expected

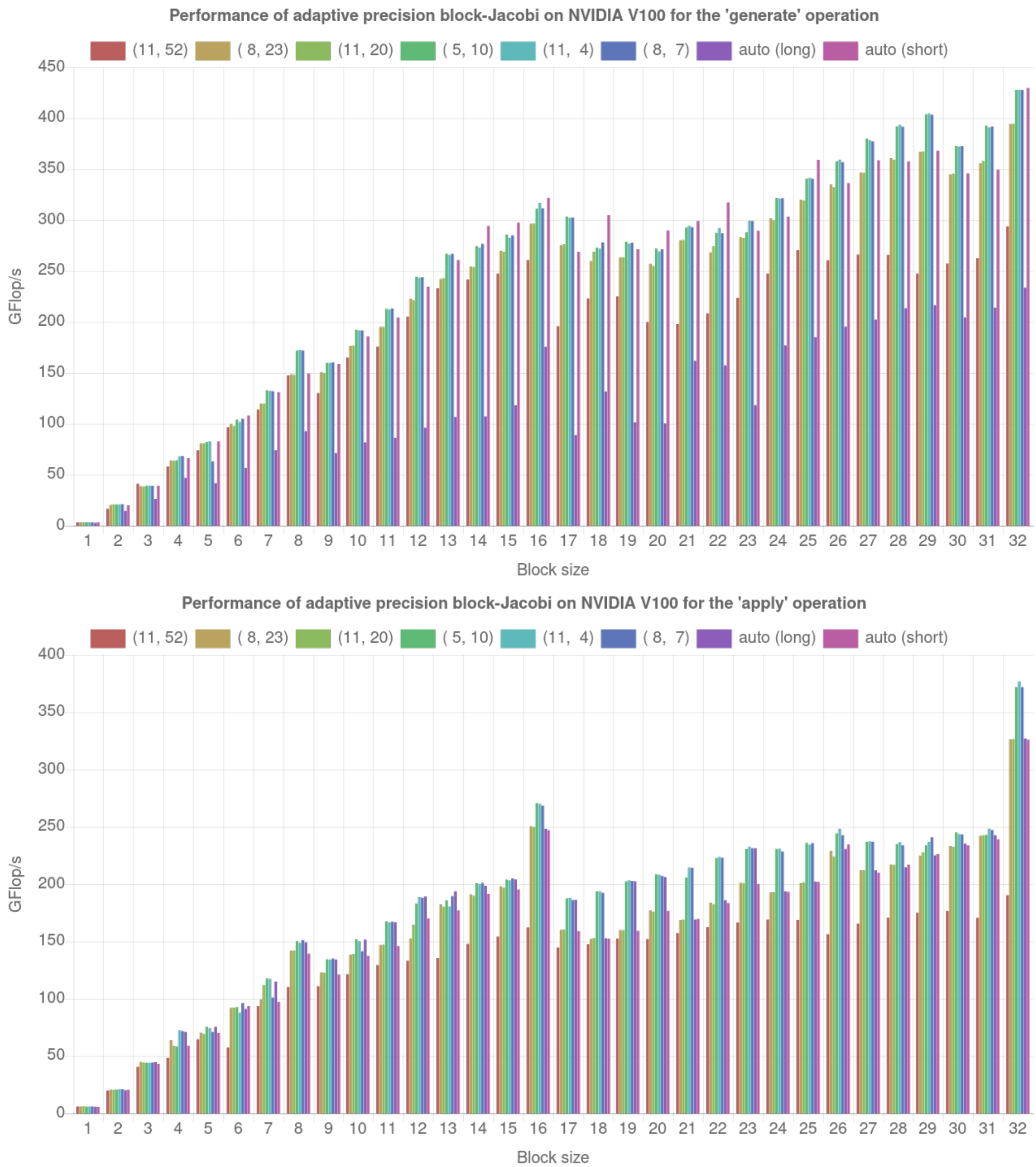

Fig. 7. Performance of adaptive precision block-Jacobi on the V100 GPU. The storage format is encoded as (*ex*, *snf*). The first bar (red), is the full precision block-Jacobi. The next five bars represent lower storage precision without accounting for the detection of the suitable precision. The last two bars include the autodetection. For the seventh bar (purple), all six precisions were considered (requiring the calculation of two additional condition numbers). In the last bar (pink), only the three precisions that do not require additional condition number calculation are considered. All performance numbers only count the floating-point operations required by the full-precision variant. The top plot shows the performance of preconditioner generation (without block size detection), and the bottom plot the performance of preconditioner application.

to converge in a few iterations (where preconditioner generation represents a high fraction of the total runtime), it may be beneficial to only use the three non-standard precisions, or to maintain the full precision block-Jacobi.

6.4 Full solver runtime

Finally, we compare the effectiveness of a solver enhanced with adaptive precision block-Jacobi with that of a solver enhanced with the full precision variant. For this experiment, we use a set of matrices from the SuiteSparse matrix collection[4], arising in real-world applications. Only symmetric positive definite problems that have between 10^6 and 5×10^8 nonzeros, and where a block-Jacobi enhanced Conjugate Gradient (CG) solver needs more than 100 iterations are considered, as these problems justify the use of a preconditioned CG solver on GPUs. The CG solver available in the Ginkgo library was employed. For both preconditioners, the double precision standard block-Jacobi as well as the adaptive precision block-Jacobi, we automatically detect natural diagonal blocks using the supervariable amalgamation algorithm. While we recognize that it may be possible to design a more advanced method of block-detection, this is still an area of active research [Goetz and Anzt 2018] that remains outside the scope of this paper. The block size upper limit was set to 32. The solvers were run for at most 10,000 iterations, or until the initial residual norm was reduced by at least 10 orders of magnitude. We used the automatic precision detection method (with all six precisions) described earlier to select the precision of each block in the adaptive precision variant. We run two parameter settings where the automatic precision detection procedure was instructed to assign precisions such that either 1 or 2 decimal digits are preserved when applying the preconditioner. This reflects the assumption that the preconditioner provides 1 and 2 digits of accuracy, respectively. For these two settings the distribution of the distinct precision formats in the preconditioner blocks is shown in Figure 8. In case of preserving 1 decimal digit of the preconditioner (top plot in Figure 8), we observe a that a significant amount of the Jacobi blocks can be be stored in less than double precision. Many blocks are stored in single or half precision, but the non-standard $fp_{8,7}$ format is also employed for a notable fraction of the Jacobi blocks. The alternative non-standard formats, $fp_{11,20}$ and $fp_{11,4}$, are irrelevant. As expected, the situation changes for the setting where we preserve 2 decimal digits of the preconditioner (bottom plot in Figure 8). Since the precision reduction is overall more conservative, no blocks are stored in the $fp_{8,7}$ format.

Figure 9 shows the iteration count and runtime of the CG solver integrated with either the full or the adaptive precision block-Jacobi preconditioner. For the adaptive precision block-Jacobi we again consider two settings where 1 digit (top plot) and 2 digits (bottom plot) of the preconditioner are preserved, respectively. A first observation is that CG enhanced with any of the variants converged for all problems (black and gray dots on top of the plot). Furthermore, the benefit of adaptive precision highly depends on the problem and the parameter setting. If most of the blocks are relatively well conditioned, the majority of the preconditioner can be stored in lower precision, yielding improvements between 10% and 30%. For problems with ill-conditioned blocks, there is no difference between the two variants, since all blocks need to be stored in full precision in order to preserve the quality of the preconditioner. In that case, it is even possible that the adaptive variant becomes slightly slower due to the additional operations needed to read and process the information about the precisions of the blocks. In particular in the setting where only 1 digit of the preconditioner is preserved, there also exist several cases were the adaptive block-Jacobi preconditioning fails to preserve the effectiveness of the preconditioner (i.e. the preconditioner is of higher quality than one digit), which results in an increase in the number of iterations which

[4]https://sparse.tamu.edu/

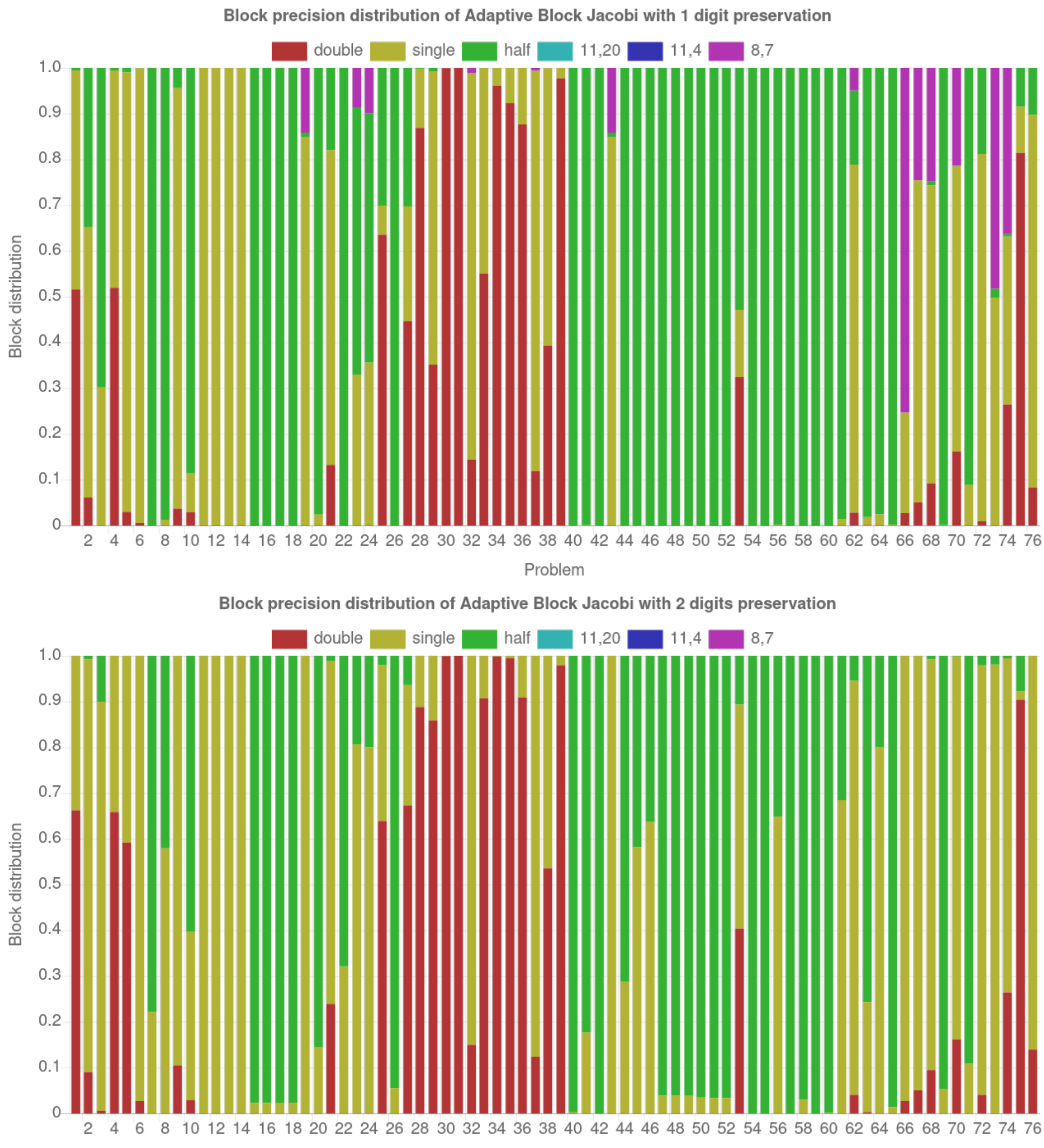

Fig. 8. Distribution of floating point formats among the distinct blocks when preserving 1 (top) or 2 (bottom) digits of the preconditioner blocks. Each column represents one of the selected matrices. The test matrices are numbered from 1 to 76. For the matrices characteristics, see Table 1. The fraction of the column filled with a certain color depicts the fraction of blocks stored in the format represented by that color.

the adaptive variant needs to converge (top plot in Figure 9). This effect is mitigated if 2 digits of the preconditioner are preserved (bottom plot in Figure 9). Furthermore, we observe that there are only few cases where the preconditioner carries more accuracy than two orders of magnitude. At the same time, the benefits of the adaptive precision block-Jacobi preserving 2 digits over the

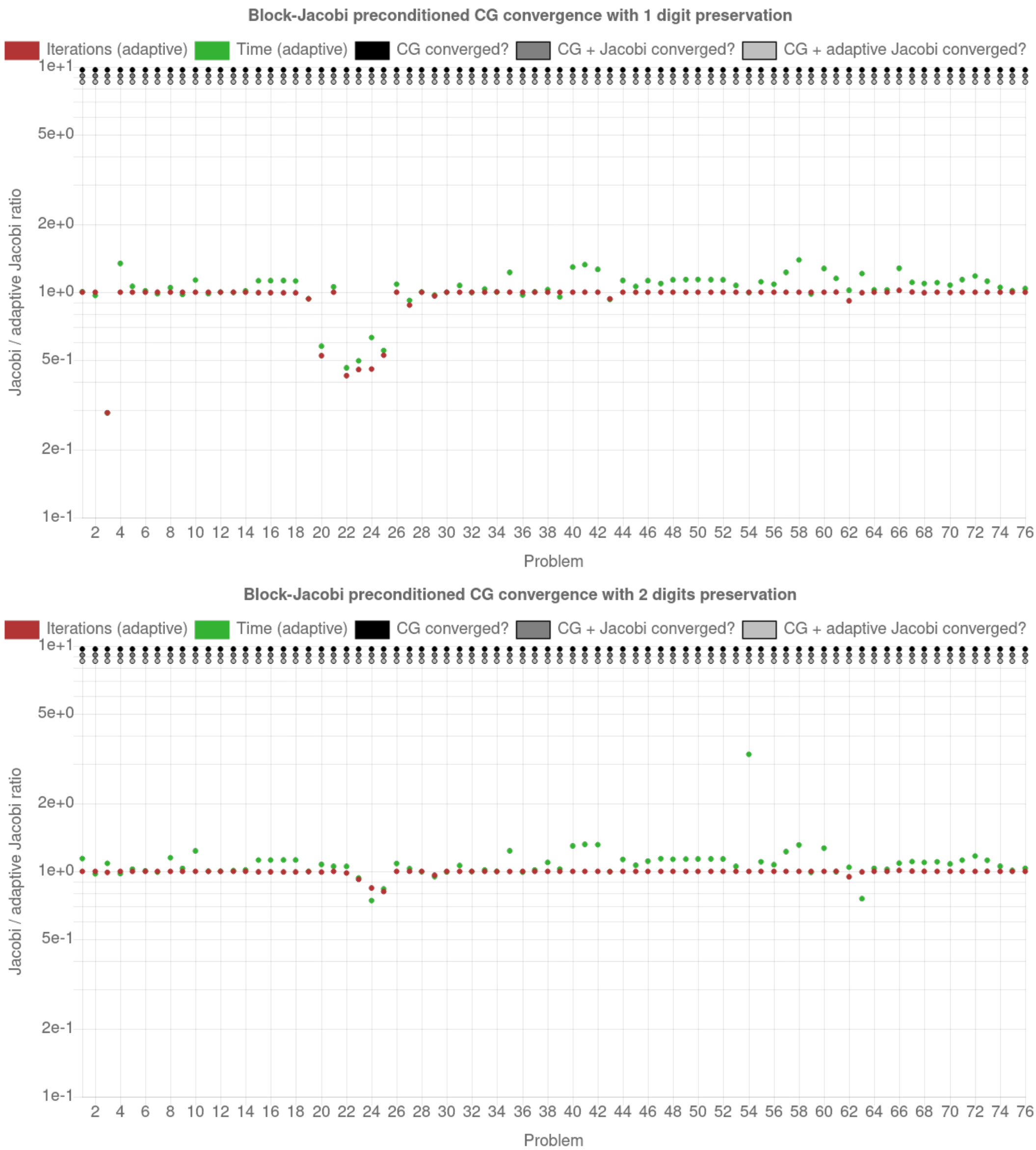

Fig. 9. Iteration count and runtime of the Conjugate Gradient (CG) solver enhanced with the adaptive precision block-Jacobi preconditioner relative to the CG solver with the full precision variant of block-Jacobi. The results include all symmetric positive definite matrices with at least 10^6 nonzeros from the SuiteSparse matrix collection for which a CG solver needs at least 100 iterations to converge. The test matrices are numbered from 1 to 76. For the matrices characteristics, see Table 1. Black and gray dots on top of the plots represent (from top to bottom) whether CG, CG+(full precision) block-Jacobi and CG+adaptive block-Jacobi converged for that matrix. The absence of a dot means that the method did not converge. The red dots represent the relative number of iterations, while the green dots the relative time of adaptive block-Jacobi compared with the full-precision variant. A value greater than 1 means that the adaptive variant outperforms the full precision block-Jacobi for that specific problem. The adaptive precision preserves 1 digit (top) or two digits (bottom) of the full precision block-Jacobi preconditioner.

standard double precision block-Jacobi are only marginally smaller than for the more aggressive setting preserving only one digit.

From this analysis, we may conclude that a setting preserving 2 digits of the preconditioner provides a good default choice, while problem-specific optimization can enable performance advantages.

7 DISCUSSION

In this section, we provide a concise discussion of the central numerical aspects coming with the precision adaptation in general, and the setting preserving 2 decimal digits of the preconditioner in particular:

(1) **Do we need a flexible Krylov solver if the preconditioner matrix is stored in lower than working precision?** No, storing the preconditioner in adaptive (lower) precision is independent of the need for a method accepting non-constant preconditioners. As elaborated in [Anzt et al. 2019], the preconditioner operator is constant as long as all arithmetic operations are handled in working precision.

(2) **Can the adaptive precision block-Jacobi matrix become singular?** No, the automatic precision adaption scheme strictly preserves the regularity of the preconditioner matrix.

(3) **Can the default setting preserving 2 decimal digits of the preconditioner introduce an iteration overhead to the outer solver?** Yes, it is possible that the block-Jacobi preconditioner has higher accuracy than 2 decimal digits. In the extreme case of the system matrix decomposing into independent problems of size smaller than the upper limit for the Jacobi blocks, the preconditioner presents the exact inverse of the system matrix and any format reduction introduces an accuracy loss. However, our analysis suggests that the block-Jacobi preconditioner rarely exceeds 2 decimal digits.

(4) **Are larger runtime savings possible by reducing the memory precision format more aggressively?** Yes, as the results in Figure 9 (top) indicate, preserving only 1 digit of the preconditioner (and therewith reducing the precision format more aggressively) can potentially augment the runtime savings for moderately-accurate block-Jacobi preconditioners. However, preserving only 1 digit in general increases the chance of loosing some preconditioner quality, and therewith increasing the iteration count.

(5) **Is it possible to control how many digits of the preconditioner are preserved?** Yes, the implementation allows to control the number of preserved preconditioner digits via a parameter.

(6) **Is the source code of the adaptive precision block-Jacobi preconditioner publicly available?** Yes, the adaptive precision block-Jacobi preconditioner is part of the Ginkgo open source software package[5]. A descriptive example for the use of the precision optimization in block-Jacobi is given in Ginkgo's adaptiveprecision-blockjacobi example[6].

(7) **Can the adaptive precision block-Jacobi preconditioner be used inside other Krylov-type solvers?** Yes, the adaptive block-Jacobi preconditioner is independent of the Conjugate Gradient method used in this work and can be employed by any solver that is amenable for preconditioning.

(8) **Can the adaptive precision block-Jacobi preconditioner be used for non-symmetric positive definite problems?** Yes, the adaptive block-Jacobi preconditioner generation is based on Gauss-Jordan elimination enhanced with pivoting [Anzt et al. 2018] and can handle general non-singular problems.

[5]https://ginkgo-project.github.io

[6]https://github.com/ginkgo-project/ginkgo/tree/develop/examples/adaptiveprecision-blockjacobi

8 CONCLUSION AND OUTLOOK

In this work we presented the first practical implementation of an adaptive precision block-Jacobi preconditioner. More precisely, we developed a heavily-tuned GPU implementation of the adaptive precision block-Jacobi preconditioner inside the Ginkgo numerical linear algebra library and made it available alongside with descriptive examples. In addition, we augmented the original strategy, which advocates for decoupling the arithmetic precision from memory precision and storing the inverted diagonal blocks in lower precision, with customized precision formats that accommodate more aggressive memory transfer savings than those that were possible with the original description of the adaptive block-Jacobi scheme. In the experimental evaluation with the adaptive precision block-Jacobi preconditioner inside a CG iterative solver, we demonstrated runtime savings between 10% and 30% compared to a full precision block-Jacobi preconditioner. The actual savings highly depend on the numerical properties of the problem, and fine-tuning the parameter controlling the level of preconditioner accuracy that is preserved may allow for even larger improvements.

In the future we plan to turn our attention to related topics such as the efficient detection of strongly connected unknowns in the system matrix and optimizing the block pattern with respect to preconditioner accuracy and memory savings.

ACKNOWLEDGMENTS

H. Anzt and T. Cojean were 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 TIN2017-82972-R of the MINECO and FEDER and the H2020 EU FETHPC Project 732631 "OPRECOMP". This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors want to acknowledge the access to the Piz Daint supercomputer at the Swiss National Supercomputing Centre (CSCS) granted under the project #d100 and the Summit supercomputer at the Oak Ridge National Lab (ORNL).

REFERENCES

Ahmad Abdelfattah, Hartwig Anzt, Erik Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Gruetzmacher, Nicholas J. Higham, Sherry Li, Neil Lindquist, Yang Liu, Jennifer Loe, Piotr Luszczek, Pratik Nayak, Sri Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung Tsai, Ichitaro Yamazaki, and Urike Meier Yang. 2020. *A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic.* SLATE Working Notes 15, ICL-UT-20-08.

Hartwig Anzt, Yen-Chen Chen, Terry Cojean, Jack Dongarra, Goran Flegar, Pratik Nayak, Enrique S. Quintana-Ortí, Yuhsiang M. Tsai, and Weichung Wang. 2019. Towards Continuous Benchmarking: An Automated Performance Evaluation Framework for High Performance Software. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '19)*. ACM, New York, NY, USA, Article 9, 11 pages. DOI: http://dx.doi.org/10.1145/3324989.3325719

Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. 2020a. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software* x, x (2020), x. DOI: http://dx.doi.org/10.21105.joss.02260

Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. 2020b. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. (2020).

Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J. Higham, and Enrique S. Quintana-Ortí. 2019. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience* 31, 6 (2019), e4460. DOI: http://dx.doi.org/10.1002/cpe.4460

Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017a. Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs. In *8th Int. Workshop Programming Models & Appl. for Multicores & Manycores (PMAM)*. 1–10.

Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017b. Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning. In *2017 46th International Conference on Parallel Processing*

(ICPP). 91–100.

Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2018. Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors. *Parallel Comput.* (jan 2018). DOI: http://dx.doi.org/10.1016/j.parco.2017.12.006

Hartwig Anzt, Jack Dongarra, Goran Flegar, Enrique S. Quintana-Ortí, and Andrés E. Tomás. 2017. Variable-Size Batched Gauss-Huard for Block-Jacobi Preconditioning. *Procedia Computer Science* 108 (2017), 1783 – 1792. DOI: http://dx.doi.org/https://doi.org/10.1016/j.procs.2017.05.186 International Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.

Hartwig Anzt, Jack Dongarra, Goran G. Flegar, and Thomas Grützmacher. 2018. Variable-Size Batched Condition Number Calculation on GPUs. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 132–139. DOI: http://dx.doi.org/10.1109/CAHPC.2018.8645907

Hartwig Anzt, Goran Flegar, Thomas Grützmacher, and Enrique S Quintana-Ortí. 2019. Toward a modular precision ecosystem for high-performance computing. *The International Journal of High Performance Computing Applications* (2019), 1094342019846547.

Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 233–244. DOI: http://dx.doi.org/10.1145/1583991.1584053

Erin Carson and Nicholas J. Higham. 2018. Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. *SIAM J. Scientific Computing* 40, 2 (2018), A817–A847. DOI: http://dx.doi.org/10.1137/17M1140819

IEEE Standard Commitee. 2000. IEEE Standard for Modeling and Simulation (M Amp;S) High Level Architecture (HLA) - Framework and Rules. *IEEE Std. 1516-2000* (2000), i –22. DOI: http://dx.doi.org/10.1109/IEEESTD.2000.92296

Siegfried Cools. 2018. Numerical stability analysis of the class of communication hiding pipelined Conjugate Gradient methods. *CoRR* abs/1804.02962 (2018). http://arxiv.org/abs/1804.02962

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI: http://dx.doi.org/10.1145/2049662.2049663

Jack Dongarra and others. 2014. *Applied mathematics research for exascale computing*. Technical Report. U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research Program. https://science.energy.gov/~/media/ascr/pdf/research/am/docs/EMWGreport.pdf.

Markus Goetz and Hartwig Anzt. 2018. Machine Learning-Aided Numerical Linear Algebra: Convolutional Neural Networks for the Efficient Preconditioner Generation. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*. 49–56. DOI: http://dx.doi.org/10.1109/ScalA.2018.00010

Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (3rd ed.). The Johns Hopkins University Press, Baltimore.

Gene H. Golub and Qiang Ye. 1999. Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iteration. *SIAM Journal on Scientific Computing* 21, 4 (1999), 1305–1320. DOI: http://dx.doi.org/10.1137/S1064827597323415

Google. https://github.com/google/googletest. (????).

Magnus R. Hestenes and Eduard Stiefel. 1952. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Standards* 49, 6 (Dec. 1952), 409–436.

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Mark Hoemmen. 2010. *Communication-avoiding Krylov Subspace Methods*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Demmel, James W. AAI3413388.

Cornelius Lanczos. 1952. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Standards* 49, 1 (Dec. 1952), 33–53.

Robert Lucas and others. 2014. Top ten Exascale research challenges. (2014). http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

NVIDIA Corporation 2018. *NVIDIA CUDA Toolkit* (9.0 ed.). NVIDIA Corporation.

Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM.

A CHARACTERISTICS OF SUITESPARSE (SS) MATRICES USED IN BENCHMARKS

						row and col nnz stats		
Number	SS id	Name	#rows	#cols	nnz	min	mean	max
1	341	bcsstk36	23052	23052	1143140	8	49.59	178
2	356	ct20stif	52329	52329	2600295	2	51.57	207
3	361	msc10848	10848	10848	1229776	45	113.36	723
4	362	msc23052	23052	23052	1142686	12	50.10	178
5	369	pwtk	217918	217918	11524432	2	53.39	180
6	761	nasasrb	54870	54870	2677324	12	48.79	276
7	804	cfd1	70656	70656	1825580	12	25.88	33
8	805	cfd2	123440	123440	3085406	8	25.02	30
9	813	olafu	16146	16146	1015156	24	62.87	89
10	817	raefsky4	19779	19779	1316789	18	67.17	177
11	936	nd3k	9000	9000	3279690	127	364.41	515
12	937	nd6k	18000	18000	6897316	130	383.18	514
13	938	nd12k	36000	36000	14220946	126	395.03	519
14	939	nd24k	72000	72000	28715634	110	398.83	520
15	942	af_shell3	504855	504855	17562051	20	34.84	40
16	943	af_shell4	504855	504855	17562051	20	34.84	40
17	946	af_shell7	504855	504855	17579155	20	34.84	40
18	947	af_shell8	504855	504855	17579155	20	34.84	40
19	1202	gyro_k	17361	17361	1021159	12	58.82	360
20	1252	audikw_1	943695	943695	77651847	21	82.28	345
21	1253	bmw7st_1	141347	141347	7318399	1	51.93	435
22	1254	bmwcra_1	148770	148770	10641602	24	71.55	351
23	1257	crankseg_1	52804	52804	10614210	48	201.01	2703
24	1258	crankseg_2	63838	63838	14148858	48	221.64	3423
25	1266	hood	220542	220542	9895422	21	48.83	77
26	1267	inline_1	503712	503712	36816170	18	73.09	843
27	1268	ldoor	952203	952203	42493817	28	48.86	77
28	1269	m_t1	97578	97578	9753570	48	99.96	237
29	1270	oilpan	73752	73752	2148558	28	48.77	70
30	1275	s3dkq4m2	90449	90449	4427725	13	53.30	54
31	1276	s3dkt3m2	90449	90449	3686223	7	41.50	42
32	1277	ship_001	34920	34920	3896496	18	133.00	438
33	1278	ship_003	121728	121728	3777036	18	66.43	144
34	1279	shipsec1	140874	140874	3568176	24	55.46	102
35	1280	shipsec5	179860	179860	4598604	12	56.23	126
36	1281	shipsec8	114919	114919	3303553	15	57.90	132
37	1283	thread	29736	29736	4444880	48	150.32	306
38	1287	vanbody	47072	47072	2329056	6	49.65	232
39	1290	x104	108384	108384	8713602	30	93.81	324
40	1403	thermal2	1228045	1228045	8580313	1	6.99	11
41	1421	G3_circuit	1585478	1585478	7660826	2	4.83	6
42	1423	apache2	715176	715176	4817870	4	6.74	8
43	1435	gyro	17361	17361	1021159	12	58.82	360
44	1453	bone010	986703	986703	47851783	12	72.63	81

45	1454	boneS01	127224	127224	5516602	12	52.78	81
46	1455	boneS10	914898	914898	40878708	12	60.63	81
47	1580	af_0_k101	503625	503625	17550675	15	34.85	35
48	1581	af_1_k101	503625	503625	17550675	15	34.85	35
49	1582	af_2_k101	503625	503625	17550675	15	34.85	35
50	1583	af_3_k101	503625	503625	17550675	15	34.85	35
51	1584	af_4_k101	503625	503625	17550675	15	34.85	35
52	1585	af_5_k101	503625	503625	17550675	15	34.85	35
53	1644	msdoor	415863	415863	19173163	28	48.67	77
54	1848	Dubcova2	65025	65025	1030225	4	15.84	25
55	1849	Dubcova3	146689	146689	3636643	9	24.79	49
56	1850	BenElechi1	245874	245874	13150496	1	53.48	54
57	1853	parabolic_fem	525825	525825	3674625	3	6.99	7
58	1883	ecology2	999999	999999	4995991	3	4.99	5
59	1892	denormal	89400	89400	1156224	6	12.93	13
60	1899	tmt_sym	726713	726713	5080961	3	6.99	9
61	1909	smt	25710	25710	3749582	52	145.98	414
62	2283	offshore	259789	259789	4242673	5	16.33	31
63	2373	pdb1HYS	36417	36417	4344765	18	119.31	204
64	2374	consph	83334	83334	6010480	1	72.13	81
65	2375	cant	62451	62451	4007383	1	64.17	78
66	2541	Serena	1391349	1391349	64131971	15	46.38	249
67	2542	Emilia_923	923136	923136	40373538	15	44.42	57
68	2543	Fault_639	638802	638802	27245944	15	44.79	318
69	2544	Flan_1565	1564794	1564794	114165372	24	75.03	81
70	2545	Geo_1438	1437960	1437960	60236322	15	43.92	57
71	2546	Hook_1498	1498023	1498023	59374451	15	40.67	93
72	2547	StocF-1465	1465137	1465137	21005389	1	14.34	189
73	2659	Bump_2911	2911419	2911419	127729899	1	43.87	195
74	2660	Queen_4147	4147110	4147110	316548962	24	79.45	81
75	2661	PFlow_742	742793	742793	37138461	1	50.00	137
76	2664	bundle_adj	513351	513351	20207907	3	39.36	12588

Table 1. Characteristics of SuiteSparse matrices used in Figures 8 and 9. All matrices are symmetric.

B REPRODUCE THE RESULTS OF THIS PAPER

In this appendix, we explain how to generate and analyze results with adaptive precision block-Jacobi, in particular, to reproduce the figure 8 from the relevant paper. We assume that the code is benchmarked on the Summit machine. If that is not the case, we cannot help with packages selection and other details such as job submission. For any issue reproducing these experiments please send a mail to mailto:ginkgo.library@gmail.com.

The main steps are as follows:

(1) Install ssget and prefetch the matrices from the SuiteSparse collection
(2) Download and build Ginkgo
(3) Prepare the experiment scripts
(4) Run the experiments
(5) Publish the experiments to github and tie to the information in the previous mail for generating the plots.

B.1 Fetching the matrices

First of all, a tool is required for benchmarking: https://github.com/ginkgo-project/ssget

This tool is a bash script simplifying downloading matrices from the SuiteSparse matrix collection. The script can be put anywhere in the PATH, but line 39 (ARCHIVE_LOCATION) has to be configured, this is where the downloaded matrices will be stored. On the Summit supercomputer, this would typically have to be somewhere in `$MEMBERWORK/<project>/....`, since this has better access inside jobs.

The matrices used for the experiments can be pre-downloaded, as this saves some node time, as is shown in Listing 1:

```
 for i in $(seq 0 $(ssget -n)); do
    posdef=$(ssget -p posdef -i $i)
    cols=$(ssget -p cols -i $i)
    nnz=$(ssget -p nonzeros -i $i)
    if [ "$posdef" -eq 1 -a "$cols" -lt 10000000 -a "$nnz" -lt 500000000 ]; then
        ssget -f -i $i
    fi
done
```

Listing 1. Download the relevant SuiteSparse matrices to reproduce the experiments.

B.2 Building Ginkgo

Afterwards, Ginkgo can be cloned, configured and built. The steps are shown in Listing 2. All paths can be adapted as needed. The <...> (project) part absolutely needs to be replaced:

```
project=<project>
ginkgo_source=$HOME/TOMS-bj-reproduce/ginkgo
ginkgo_build=$MEMBERWORK/${project,,}/TOMS-bj-reproduce/ginkgo-build
module load gcc/6.4.0 cuda/9.2.148 cmake/3.15.2 git/2.20.1
# For every new session, the previous setup is required
git clone https://github.com/ginkgo-project/ginkgo.git ${ginkgo_source} --branch 2019toms-adaptive
     -bj-solver
mkdir -p ${ginkgo_build} && cd ${ginkgo_build}
cmake -DBUILD_CUDA=on -DBUILD_OMP=off -DBUILD_EXAMPLES=off -DBUILD_GTEST=on -DDEVEL_TOOLS=off -
     DCMAKE_C_COMPILER=$(which gcc) -DCMAKE_CXX_COMPILER=$(which g++) ${ginkgo_source}
bsub -P ${project^^} -W 2:00 -nnodes 1 jsrun -n 1 -c 10 -g 0 make -j10
# This is a good time to go do something else, compilation will take a
# while as there is a big CUDA compiler bug which makes it extremely slow and
# memory heavy to # compile the block jacobi with all optimizations.
make -j10 # afterwards, ensure everything is compiled
make test
# Everything should run without failure.
```

Listing 2. Download and build the Ginkgo software to reproduce the experiments.

B.3 Prepare the experiment scripts

In Listing 3, we create two files for launching the experiments. A `ginkgo_benchmark.lsf` script for `bsub`, and a `benchmark_one_node.sh` script which runs jsrun and populates some arguments in order to create segments to be benchmarked, all of which can run in parallel.

```
cat > ${ginkgo_source}/benchmark_one_node.sh << EOF
#!/bin/bash -x

cd \${1}/benchmark
chmod +x run_all_benchmarks.sh

ADAPTIVE_JACOBI_ACCURACY=\${4:-1e-1}
export BENCHMARK=solver
export PRECONDS=none,jacobi,adaptive-jacobi
export SYSTEM_NAME=V100_summit
export SEGMENT_ID=\${2}
export SEGMENTS=\${3}
./run_all_benchmarks.sh >/dev/null
EOF

cat > $ginkgo_source/benchmark_ginkgo.lsf << EOF
#!/bin/bash
#BSUB -P ${project^^}
#BSUB -W 2:00
#BSUB -nnodes 1
#BSUB -J Ginkgo_Benchmark
#BSUB -o Ginkgo_Benchmark.%J
#BSUB -e Ginkgo_Benchmark.%J

if [ -z \${segment_id+x} ]
then
        echo "Please set variable segment_id"
        exit
fi

if [ -z \${segments+x} ]
then
        echo "Please set variable segments"
        exit
fi

module load gcc/6.4.0 cuda/9.2.148 cmake/3.15.2 git/2.20.1

jsrun -n 1 -a 1 -c 1 -g 1 $ginkgo_source/benchmark_one_node.sh $ginkgo_build \$segment_id \
     $segments
EOF

chmod +x ${ginkgo_source}/benchmark_one_node.sh
```

Listing 3. Generate the scripts required for launching the Ginkgo benchmarks

B.4 Run the benchmarks

To run the benchmarks there are two parameters to pick:

- the parallelism desired,
- the number of matrices we want to reproduce against (all of them or a portion).

These are controlled with the variables `segments` and `segment_id`. As an example, the code shown in Listing 4 will run 20 benchmarks in parallel and benchmark all matrices since we use all `segment_id`.

```
for i in $(seq 1 20); do segments=20 segment_id=$i bsub $ginkgo_source/benchmark_ginkgo.lsf; done
```

Listing 4. Benchmark Ginkgo using 20 jobs in parallel

To only benchmark the first half of the matrices, we could do like in Listing 5:

```
# Note the different in the `seq` below
for i in $(seq 1 10); do segments=20 segment_id=$i bsub $ginkgo_source/benchmark_ginkgo.lsf; done
```

Listing 5. Benchmark Ginkgo on only half the matrices

B.5 Publish the results and generate the plots

For analyzing the results, any tool can be used. The previous experiments generated json files for each matrix, each containing timing and convergence results without preconditioner, with standard block-Jacobi preconditioner, and with adaptive precision block-Jacobi.

In this section, we describe how to generate the plots by using Ginkgo's GPE[7] tool. First, we need to publish the experiments into a Github repository which will be then linked as source input to the GPE. For this, we can simply fork the ginkgo-data repository. To do so, we can go to the github repository and use the forking interface: https://github.com/ginkgo-project/ginkgo-data/tree/2019toms-adaptive-bj

Once this is done, we want to clone the 2019toms-adaptive-bj branch, move all results into a public domain, and access the GPE for plotting the results. The detailed steps are shown in Listing 6.

```
git clone https://github.com/<username>/ginkgo-data.git ${ginkgo_build}/benchmark/ginkgo-data --
    branch 2019toms-adaptive-bj
rsync -rtv ${ginkgo_build}/benchmark/results/ ${ginkgo_build}/benchmark/ginkgo-data/data/
cd ${ginkgo_build}/benchmark/ginkgo-data/data/
# The following updates the main `.json` files with the list of data
module load python/3.7.0
./build-list . > list.json
./agregate < list.json > agregate.json
git config --local user.name "<Name>"
git config --local user.email "<email>"
git commit -am "Ginkgo Reproduced BJ data"
git push
```

Listing 6. Publish the results and generate summary files to a Github benchmark repository.

For generating the plots in the GPE, here are the steps to go through:

(1) Access the GPE: https://ginkgo-project.github.io/gpe/
(2) Update data root URL, from https://raw.githubusercontent.com/ginkgo-project/ginkgo-data/master/data to https://raw.githubusercontent.com/<username>/ginkgo-data/2019toms-adaptive-bj/data
(3) Click on the arrow to load the data, select the `Result Summary` entry above. The first few entries under this should be V100 (cuda).
(4) Click on select an example to choose a plotting script, and update the url from https://raw.githubusercontent.com/ginkgo-project/ginkgo-data/master/plots to https://raw.githubusercontent.com/<username>/ginkgo-data/2019toms-adaptive-bj/plots
(5) Again Click on the arrow next to the URL to load everything
(6) Select the plot "Preconditioned CG detailed comparison"
(7) The results should be available in the tab "plot" on the right side

B.6 Generate results and plots for precision 1e-2

The previous steps benchmarked and generated the plot with block Jacobi accuracy $1e-1$, to generate the results with $1e-2$, both steps 4 and 5 need to be repeated. The only modification necessary is to edit `$ginkgo_source/benchmark_ginkgo.lsf` by appending `"1e-2"` to the end of the jsrun line.

In GPE, plotting with the previous link will now show the benchmark data of precision $1e-2$ by default. To get back to the previous $1e-1$ precision results, replace `2019toms-adaptive-bj` in the link by the previous commit hash.

[7]https://ginkgo-project.github.io/gpe/