A customized precision format based on mantissa segmentation for accelerating sparse linear algebra

Thomas Grützmacher[1] **Terry Cojean**[1] **Goran Flegar**[2] **Fritz Göbel**[1] **Hartwig Anzt**[1,3]

[1]Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Karlsruhe, Germany
[2]Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castelló, Spain
[3]Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee

Correspondence
Hartwig Anzt, Steinbuch Centre for Computing, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany; or
Innovative Computing Lab, University of Tennessee, Knoxville, TN 37996.
Email: hartwig.anzt@kit.edu

Funding information
Helmholtz Association, Grant/Award Number: VH-NG-1241; CICYT, Grant/Award Number: TIN2017-82972-R; Swiss National Supercomputing Centre, Grant/Award Number: #d100; Juelich Supercomputing Centre

Summary

In this work, we pursue the idea of radically decoupling the floating point format used for arithmetic operations from the format used to store the data in memory. We complement this idea with a customized precision memory format derived by splitting the mantissa (significand) of standard IEEE formats into segments, such that values can be accessed faster if lower accuracy is acceptable. Combined with precision-aware algorithms that dynamically adapt the data access accuracy to the numerical requirements, the customized precision memory format can render attractive runtime savings without impacting the memory footprint of the data or the accuracy of the final result. In an experimental analysis using the adaptive precision Jacobi method on diagonalizable test problems, we assess the benefits of the mantissa-segmenting customized precision format on recent multi- and manycore architectures.

KEYWORDS

adaptive precision Jacobi, GPUs, mantissa segmentation, modular precision ecosystem, multiprecision algorithm, multicore

1 INTRODUCTION

The significantly slower memory bandwidth growth, compared to the growth of arithmetic performance of the processing units, poses a roadblock for the continuation of the success of digital processing from supercomputers to mobile devices like smartphones and wearables. The memory bottleneck for many scientific computations results in applications utilizing only a fraction of the computational power. At the same time, the memory operations are the primary energy consumer of modern architectures, heavily impacting the resource cost of applications and severely limiting the battery life of mobile devices. Without technical advances on the horizon, a disruptive paradigm shift with respect to how data is stored and processed is necessary to prepare scientific computing algorithms for the Exascale computing era. To that end, we propose to (a) pursue the idea of radically decoupling the data storage format from the processing format, (b) design a “modular precision ecosystem” that accommodates more flexibility in terms of customized data formats and memory access, and (c) develop innovative algorithms and applications that dynamically adapt data access accuracy to the numerical requirements.

In this work, we continue the research effort on a “customized precision format based on mantissa segmentation (CPMS)” that was initially presented in the work of Grützmacher and Anzt.[1] The CPMS concept allows the dynamic adaptation of the accuracy at which data is accessed in main memory. To that end, we decompose the significand (colloquially, mantissa) of the standard IEEE floating point formats into segments, which enables accessing only part of the significand information. Taking the standard IEEE floating point formats as basis allows to quickly obtain a standard representation of the value by filling the potentially omitted significand segments with default values. Once a standard representation is obtained, arithmetic operations are performed using the built-in hardware support for IEEE arithmetic of modern processors. Without impacting the precision of arithmetic operations, this significantly reduces the cost of data access if lower accuracy is acceptable.

In our previous work,[1] we presented the potential of this strategy by modifying a simple Jacobi relaxation method to adapt the memory access to the numerical requirements. The underlying idea, detailed in Section 5, is to derive an adaptive-precision Jacobi method[2] that modulates the memory access precision on a component level to the convergence of the iterative process. The data access kernels of the CPMS format have to reflect the characteristics of the underlying hardware architecture. Here, we extend the efforts presented in our previous work[1] by redesigning the GPU backend for `CPMS`, and proposing a backend for multicore architectures supporting vectorization. A comprehensive experimental evaluation in Section 6 reveals attractive runtime savings compared to a standard Jacobi relaxation on state-of-the-art HPC architectures. We conclude in Section 7 with an outlook on future work. For convenience, we start in Section 2 with a brief overview about mixed precision linear algebra strategies, and provide some background about the Jacobi relaxation method in Section 3. In Section 4, we motivate the mantissa adaptation strategy for diagonalizable iteration matrices by referring to the matrix power iteration and its property of the largest eigenvalue becoming dominant after few iterations.

We note that, in this paper, we interchangeably use the terms "mantissa" and "significand" to bridge between the research presented in our previous work,[1] and a reviewer comment on an earlier version of the paper that points out the use of the term "mantissa" is discouraged by the IEEE standard committee.[3]*

2 RELATED WORK ON MIXED PRECISION NUMERICS

In digital processing, the cost of communication, memory access, and arithmetic operations typically correlates with the complexity of the precision format used,[4] ie, the more bits are involved, the higher is the cost for reading, writing data to memory, communicating between processors, or performing operations. For scientific simulation codes and many applications in data analytics, the IEEE 754 double precision format (`fp64`) is established as the de-facto standard. Nevertheless, there exist efforts to combine double precision with lower precision formats with the intention to reduce the resource footprint while preserving the quality of the algorithms' output. In numerical linear algebra, the accuracy of the result usually correlates with the conditioning of the problem and the floating point format that is employed to represent the values. Numerical effects, such as rounding errors, can result in a less accurate solution or even breakdown if a "lower precision format" (in terms of less significand and exponent bits) is used. An intuitive example are fixed-point iterations like the Jacobi[5] relaxation, ie, if a linear system fulfills the convergence condition for the Jacobi relaxation, the iterations converge toward a solution with the approximation accuracy correlating with the employed precision format. If the precision format is extended, ie, by allowing for more accuracy in terms of mantissa bits, the iterations will stagnate later, increasing the accuracy of the solution approximation.[2]

There exist different strategies to exploit this property. One approach is to embed an iterative solver running in lower precision inside an iterative refinement method using working precision. In this mixed precision iterative refinement (MPIR[6-8]), the algorithm solving the residual system and carrying the lion's share of the workload employs a precision format that is cheaper than working precision, while the iterative refinement ensures the quality of the result. A different strategy initially starts the iterative process in lower precision, and then dynamically extends the precision format to enable convergence to double precision accuracy.[2] What most existing mixed precision strategies share is a tight coupling between the "arithmetic precision format" used for arithmetic operations, and the "storage format" that is used for handling the data in memory. While this choice seems to be intuitive, it ignores the hardware trend of computational power growing at a much faster pace than memory bandwidth.[9]

In a pioneering proposal to break up this coupling,[10] the authors assessed the resource savings obtained by storing a double precision block-Jacobi preconditioner in a lower precision format. Given that the preconditioner stays constant in memory, this is a preliminary step towards the idea of decoupling the memory format from the arithmetic format for a complete iterative solution process. In an earlier publication,[1] we developed an iterative solver that decouples the memory format from the arithmetic format in all steps of the algorithm. The concept is based on the adaptive precision Jacobi[2] method that dynamically adapts the precision format to the numerical requirements. The convergence of the Jacobi iterations with a constant relaxation factor is exploited by quickly reacting to variations in the convergence rate and changing the accuracy of the storage format.[2] This strategy is realized by decomposing the IEEE standard precision formats used in the arithmetic operations into segments, and accessing only a subset of the significand segments when reading data from memory if the numerical properties allow for it. The use of customized memory formats instead of lower precision IEEE formats like single precision or half precision efficiently removes the need for protecting against under- and overflow or duplicating data in memory.[1] While that implementation is realized on a high-end GPU architecture, here, we extend the scope of the adaptive precision Jacobi using the customized precision format based on mantissa segmentation ("`CPMS`") to general purpose CPU architectures. This is more challenging than the GPU implementation, as, unlike the CUDA compiler, the multicore compilers have to extract parallelism from the sequential programming model in order to properly vectorize the code. Before providing details about how `CPMS` is realized on CPU and GPU architectures, we first recall the adaptive Jacobi method serving as showcase algorithm that can efficiently exploit the modular precision ecosystem.

*We highly appreciate the reviewer's comment and will take it into account in future presentation of research material.

3 JACOBI ITERATION

The Jacobi relaxation[5] is a method for the iterative solution of linear systems of the form $Ax = b$, $A \in \mathbb{R}^{n \times n}, b, x \in \mathbb{R}^n$. Basis for the Jacobi method is the reformulation of the linear system $Ax = b$ as

$$\begin{aligned} Ax &= b \\ \Leftrightarrow (L + D + U)x &= b \\ \Leftrightarrow (L + U)x + Dx &= b \\ \Leftrightarrow x &= D^{-1}b - D^{-1}(L + U)x. \end{aligned}$$

Thus, matrix splitting $A = L + D + U$ allows to derive component-parallel updates for the vector of unknowns of the form

$$\begin{aligned} x^{\{k\}} &:= D^{-1}\left(b + (D - A)x^{\{k-1\}}\right) \\ &= D^{-1}b + Mx^{\{k-1\}}, \qquad k = 1, 2, \dots, \end{aligned} \tag{1}$$

where $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix that only contains the diagonal entries of A, and $x^{\{0\}}$ corresponds to a starting solution guess.[5] For a regular linear system of equations, the Jacobi iteration is guaranteed to converge to the unique solution if the spectral radius of the iteration matrix $M = (I - D^{-1}A)$ is smaller than one.[5]

4 COMPONENT-WISE CONTRACTION

For an (iteration) matrix with spectral radius smaller one, the eigenvalues $\lambda_1 \dots \lambda_n$ of the corresponding eigenvectors $\xi_1 \dots \xi_n$ are located inside the unit disc around 0. This implies that the linear operator $\mathcal{L} : x \to Mx$ induced by the matrix is for all eigenvectors contractive in every component. If the matrix M is a diagonalizable matrix, all eigenvectors are pairwise distinct. In this case, the eigenvectors span the complete space $\mathbb{R}^n$, and any vector $x \in \mathbb{R}^n$ can be expressed as a linear combination of the eigenvectors

$$x = \mu_1\xi_1 + \cdots + \mu_n\xi_n, \quad \mu_i \in \mathbb{R}.$$

Examples for diagonalizable matrices are involutions, projections, and symmetric matrices.

Applying the linear operator $\mathcal{L}$ induced by a diagonalizable matrix to x gives

$$Mx = \sum_{m=1}^{n}(M\mu_m\xi_m) = \sum_{m=1}^{n}\mu_m\lambda_m\xi_m \tag{2}$$

with $|\lambda_i| \in [0, 1) \forall i = 1 \dots n$. We assume that the Eigenvalues of M can be ordered such that there exists $r \leq n$ with

$$\begin{aligned} &\lambda_1 = \dots = \lambda_r \\ &|\lambda_1| = \dots = |\lambda_r| > |\lambda_{r+1}| \geq \dots \geq |\lambda_n| \end{aligned}$$

and that the initial vector x comprises a nonzero contribution in the direction of an eigenvector corresponding to the largest eigenvalue (in absolute value), ie, $x = \sum_m \mu_m\xi_m$, where $\mu_i \neq 0$ for at least one $i \in \{1, \dots, r\}$. Then, the repetitive application of the operator $\mathcal{L}$ emphasizes the contribution of the largest eigenvalue, and, in case the input is normalized before every application of the linear operator, enforces the convergence towards this eigenvector (or linear combination of eigenvectors). This principle is the basis for the Matrix Power Iteration,[11] an effective iterative method for finding the dominant eigenvalue/eigenvector pair.

We can rewrite (2) into a composition of standard basis vectors

$$\begin{aligned} Mx &= \mu_1\lambda_1\xi_1 + \dots + \mu_n\lambda_n\xi_n \\ &= (\alpha_1\lambda_1 + \beta_1\lambda_2 + \gamma_1\lambda_3 + \dots)e_1 + \dots + (\alpha_n\lambda_1 + \beta_n\lambda_2 + \gamma_n\lambda_3 \dots)e_n. \end{aligned} \tag{3}$$

Obviously, repetitive application of the linear operator $\mathcal{L}$ to x then gives

$$\begin{aligned} M^k x &= \mu_1\lambda_1^k\xi_1 + \cdots + \mu_n\lambda_n^k\xi_n \\ &= \left(\alpha_1\lambda_1^k + \beta_1\lambda_2^k + \gamma_1\lambda_3^k + \dots\right)e_1 + \dots + \left(\alpha_n\lambda_1^k + \beta_n\lambda_2^k + \gamma_n\lambda_3^k \dots\right)e_n. \end{aligned} \tag{4}$$

In practice, we observe that a moderate value for k makes the contr butions related to the largest eigenvalue λ_1 dominant. This motivates for values k larger some (moderate) threshold k_t to ignore the (increasingly irrelevant) contributions coming from the smaller eigenvalues, and use the approximation

$$M^k x = \lambda_1^k \sum_i \alpha_i e_i + \varepsilon \quad \forall k \geq k_t \tag{5}$$

with $|\varepsilon|$ decreasing for increasing k.

Obviously, (5) implies that the linear operator is contracting all components equally with a constant contraction factor, which is equivalent to the largest eigenvalue λ_1. In consequence, for the induced Jacobi iteration, we get that, for every component i, we have

$$\left|x_i^{\{k+3\}} - x_i^{\{k+2\}}\right| = \theta \left|x_i^{\{k+2\}} - x_i^{\{k+1\}}\right| = \theta^2 \left|x_i^{\{k+1\}} - x_i^{\{k\}}\right| \dots, \tag{6}$$

and thus

$$c := \frac{1}{\theta} = \frac{z_i^{\{k\}}}{z_i^{\{k+1\}}} = \frac{\left|x_i^{\{k+1\}} - x_i^{\{k\}}\right|}{\left|x_i^{\{k+2\}} - x_i^{\{k+1\}}\right|}, \quad k \geq k_t. \tag{7}$$

We note that this component-constant contraction property strictly only holds if the initial guess x_0 is a multiple of the eigenvector corresponding to the largest eigenvalue. It also holds (for a different contraction coefficient) in situations where the initial guess is a different eigenvector of the linear operator. In practice, the initial guess usually comprises contributions from all eigenvectors. At the same time, we observe that the limited precision induced by the use of floating point formats results in a setting where, after few iterations (moderate k), the contributions of the smaller eigenvalues are covered by the noise of the arithmetic rounding effects.

These observations motivate to design a strategy that is based on monitoring the contraction factor on a component level to retrieve information about the convergence and stagnation of the Jacobi iterations. Specifically, the idea is to detect stagnation by monitoring $z_i^{\{k\}}$ at component level with some periodicity ϕ, and use the stagnation test

$$\left|\frac{z_i^{\{k\}}}{z_i^{\{k+\phi\}}} - c^\phi\right| > \tilde{\delta} \tag{8}$$

with some threshold $\tilde{\delta}$. Once the threshold is exceeded, the memory routines start accessing additional mantissa information to increase the accuracy and avoid early stagnation of the Jacobi iterations prior to reaching convergence in double precision accuracy. As elaborated, not only the detection test but also the modulation of the significand length can happen on a component level. This, however, requires keeping track of a set of $z_i^{\{k\}}$ entries and reading the precision format configuration flag for every component. Alternatively, the significand adaptation can be realized on a global level, synchronizing the memory access precision across all components and efficiently reducing the access cost to the format accuracy information. An intermediate strategy is to modulate significand length on the block level (eg, using the same precision for an entire cache line). However, our experiments indicated that this does not result in statistically relevant performance improvements over the component-level version.

While the test periodicity ϕ and the stagnation test threshold $\tilde{\delta}$ can be optimized for each problem individually, we use the values $\tilde{\delta} = 0.9 \cdot (c^\phi - 1)$ and $\phi = 10$ as they offer good balance between robustness and performance.[1] Obviously, k_t has to be chosen carefully with respect to how quickly the largest eigenvalue becomes dominant. For the specific tests in this work, the choice $k_t = 10$ facilitated good results, but we acknowledge that larger values for k_t may be necessary when addressing more challenging problems.

5 CUSTOMIZED PRECISION FORMAT BASED ON MANTISSA SEGMENTATION

The efficient use of the mantissa-segmenting customized precision format combines the following central ideas[1]: (1) the memory format is completely decoupled from the arithmetic format, and (2) instead of using the standard IEEE floating point formats in memory operations, the values are split into segments and stored separately. This allows reading only a part of significand information if the algorithm properties allow it. While these two ideas can be addressed independently, they work especially well when used in combination.

Decoupling the arithmetic format from the storage format is motivated by low arithmetic intensity of many linear algebra routines. If the algorithm can accept reading/writing values with less accuracy, the data can be accessed much faster in a lower precision format. The arithmetic operations can still use high precision without impacting the performance as long as the algorithm remains memory bandwidth bound.[1]

Dynamically adapting the storage format to accuracy demands generally requires the duplication of data in different precision formats. This is particularly true if one relies on the standard IEEE floating point formats, as these differ in terms of how many bits are devoted to store exponent and significand information, respectively. In the CPMS strategy that we initially proposed in our previous work,[1] we break out of the IEEE format ecosystem by splitting a baseline format into segments and modulating the complexity of the customized precision by configuring the access to the distinct significand segments. This way, the accuracy of the values (and the cost of accessing them) depends on whether more or less significand segments are included. Acknowledging that this strategy may come with the drawback of a high number of bits being used for the exponent (particularly in comparison to the IEEE standard that decreases the number of exponent bits with the format's complexity), the

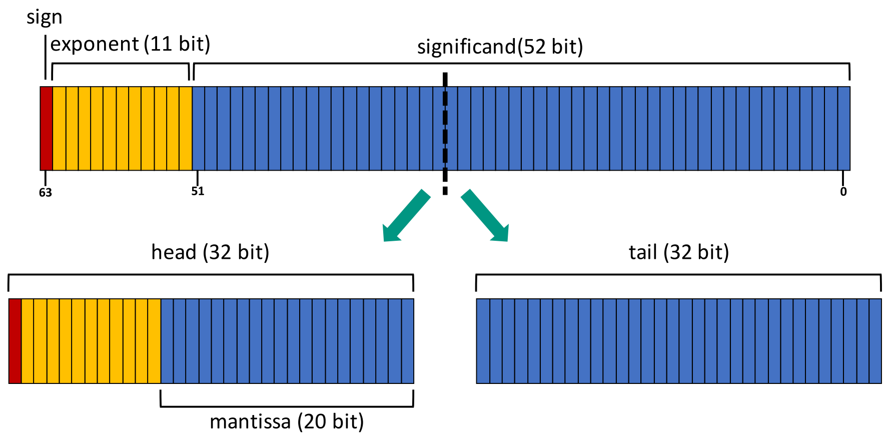


FIGURE 1 Splitting an IEEE double precision number into "head" and "tail"[1]

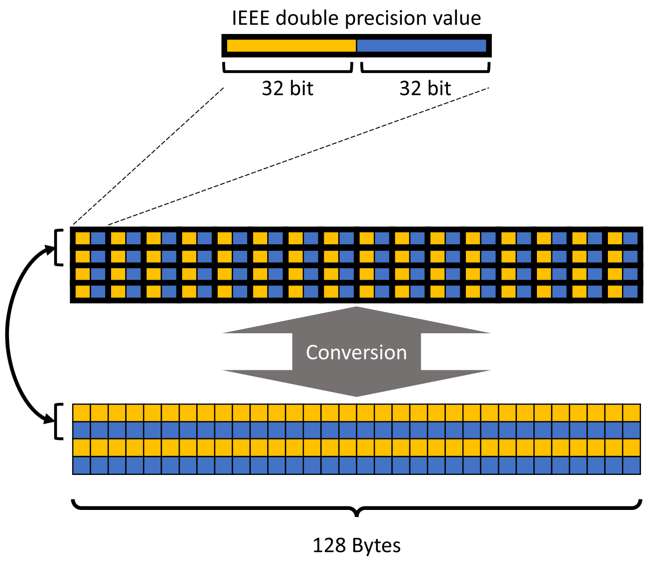


FIGURE 2 Storing the segments of a 2-segment splitting in interleaved fashion for efficient 128-byte access

approach is attractive as it (1) removes the need to duplicate data in different precision formats, (2) removes the need to meticulously protect against under- and overflow, and (3) allows efficient conversion to baseline precision by filling omitted significand bits with zeros.

If an algorithm modulates the memory accuracy over the execution time, it is necessary to accompany the `CPMS` format with a format specification information. This is realized using a byte-sized flag per value, which specifies the amount of segments that have to be read.

Since virtually all processor architectures include hardware support for the standard IEEE floating point formats, we take IEEE double precision `fp64` as the baseline precision format for `CPMS`. We point out that by preserving the exponent bits of `fp64`, the segmentation in `CPMS` can not turn a valid number into "NaN" or infinity, as both are defined by all exponent bits being filled with "1 bits".[3]

In Figure 1, we visualize the `CPMS` strategy for a 2-segment splitting of `fp64`. For this specific decomposition, we refer to the two 32-bit segments as "head" and "tail" of the customized precision format and note that the first 32 bits include less significand bits than the 32-bit long IEEE single precision format.[3] While arbitrary splittings are theoretically possible, it is beneficial if the splitting configuration aligns with the hardware-specific data access characteristics, such as efficient access to 16-bit words. In the experiment section, we evaluate a 2-segment `CPMS` and a 4-segment `CPMS` using `fp64` as baseline format.

Independently of the segment configuration, a key to the efficient use of `CPMS` is to design architecture-aware memory layouts and data access routines. While this seems to be an implementation detail, the efficient access to the values with distinct accuracy is performance-crucial on streaming architectures like GPUs where each memory read accesses 128 bytes of contiguous memory, as well as CPUs where memory access is optimized for the standard IEEE floating point formats. As a result, the `CPMS` realizations are extremely architecture-dependent, and depending on the system, the data has to be rearranged in memory before invoking `CPMS` routines.

5.1 Mantissa-segmenting customized precision on GPUs

On GPUs, each memory read commonly accesses 128 bytes of contiguous memory, and utilizing only part of the data inevitably results in low performance. This means that accessing only a subset of the significand segments requires those segments to be adjacent in memory, and not

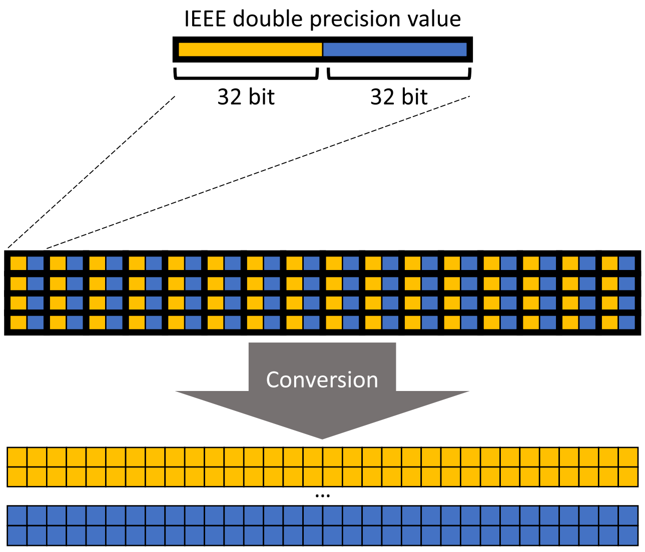


FIGURE 3 Storing the segments of a 2-segment splitting in separate memory blocks

```
// Convert function for reading in CPMS
// and returning IEEE double precision format
__device__ __forceinline__ double mergeToDouble(int32_t head, int32_t tail)
{
    double val;
    int32_t *halfVal = reinterpret_cast<int32_t *>(&val);
    halfVal[1] = head;
    halfVal[0] = tail;
    return val;
}
```

FIGURE 4 Pseudocode of the CPMS access routines for GPUs

interleaved with other (unused) significand segments. One strategy to reflect this situation is to arrange the data in 128-byte memory blocks containing the same segments (eg, 128 bytes of "heads" followed by 128 bytes of "tails" like in Figure 2).

An alternative is to completely separate the segments into two memory blocks, such as shown in Figure 3. Acknowledging both strategies have advantages and provide similar performance, we implement the second strategy completely separating the segments into disjoint memory blocks.

The CPMS access kernels we design for GPUs reassemble the arithmetic precision format from the distinct segments using `reinterpret_cast`, as shown in Figure 4. On this architecture, such implementation realizes all conversions in registers, avoiding the use of performance-detrimental shared memory.

5.2 Mantissa-segmenting customized precision on CPUs

On the CPU, every data read fetches a full line of data into the cache (similarly to the GPU). The cache line size is processor-specific; most architectures use cache line sizes of 64 bytes. Therefore, to ensure good bandwidth utilization, a strategy similar to the approach in the GPU version is employed. In particular, it is essential to access the data via vector instructions and loop parallelism. In consequence, also for the CPU realization, it is important to store the data segments adjacent in memory, and to separate the "heads" from the "tails," as shown in Figure 3.

The format conversion necessary to generate the IEEE double precision format for the arithmetic operations can be implemented following the same strategy like in the GPU version. However, experiments revealed that the use of `reinterpret_cast` results in very poor performance on the CPU. For this reason, it is necessary to follow a different strategy to provide the same functionality at higher efficiency. To that end, we employ a `union` between `double` and `int64_t` (making use of type punning[†]) and masking operations on the integer part to implement the conversion. Nevertheless, due to poor automatic vectorization support with this version (maybe due to the read-after-write dependency adjacent to the union), it is necessary to vectorize this approach manually using Intel intrinsics. In Figure 5, we illustrate the use of `union` for the conversion and a data access kernel manually vectorized using AVX2. To leverage multicore parallelism and further improve the algorithm's performance, we parallelized all loops using OpenMP directives.

6 EXPERIMENTAL EVALUATION

To show that the concept of CPMS renders benefits across a wide range of architecture designs, we run the experimental evaluation on two complementary high-end processing units, ie, on an Intel Skylake CPU and an NVIDIA V100 GPU. These architectures are fundamentally

[†]We note that type punning with unions is legal in the C11 standard, as clarified by footnote 95. To our knowledge, this is not clear when using C++, but community compilers such as GNU Compilers are supporting it.

```
// union version - not vectorized
union converterUnion {
    double dbl;
    int64_t it;
};

double readCpms(uint32_t *head, uint32_t *tail, int64_t index)
{
    converterUnion target;
    target.dbl = head[index];
    target.it = target.it << 32;
    target.it = target.it | tail[index];
    return target.dbl;
}

// vectorized version using intrinsics
__m256d readPack(const int32_t *head, const int32_t *tail, int startIndex)
{
    __m128i tailPack = _mm_loadu_si128((__m128i *)(tail + startIndex));
    __m256i exTailPack = _mm256_cvtepu32_epi64(tailPack);

    __m128i headPack = _mm_loadu_si128((__m128i *)(head + startIndex));
    __m256i exHeadPack = _mm256_cvtepu32_epi64(headPack);
    exHeadPack = _mm256_slli_epi64(exHeadPack, 32);
    //the cast has zero overhead, just there to make compiler happy
    return _mm256_castsi256_pd(_mm256_or_si256(exHeadPack, exTailPack));
}

//returns four double values, which are read in original IEEE format
__m256d readPack(const double *dblPointer, int startIndex)
{
    return _mm256_loadu_pd(dblPointer+startIndex);
}
```

FIGURE 5 Pseudocode of the CPMS access routines for CPUs

TABLE 1 Key characteristics of the Intel Xeon Gold 6148 CPU and the NVIDIA V100 GPU

	Intel Xeon Gold 6148	NVIDIA V100
Architecture	Skylake	Volta
DP performance	1.5 TFLOPs	7 TFLOPs
SP performance	3.0 TFLOPs	14 TFLOPs
Cores/SMs	20	80
Operating freq.	2.40 GHz (3.70 GHz Boost)	1.53 GHz
Mem. capacity	192 GByte	16 GByte
Mem. bandwidth	119 GByte/s	900 GByte/s
L3 cache size	27.5 MB	-
L2 cache size	20 MB	6 MB
L1 cache size	1.25 MB	128 KB

different in design and operating scheme, which makes a comparison of the hardware specifics difficult. For convenience, in Table 1, we list some characteristics of the Intel Xeon Gold 6148 CPU and the NVIDIA V100 GPU as they are configured in the JUWELS system of the Juelich Supercomputing Centre.[‡] We emphasize, however, that "cores/SMs" and "Memory Bandwidth" have a very different meaning for the distinct architectures.

The GPU version was implemented using the CUDA programming model[12] and compiled with the nvcc compiler from the CUDA 9.2 toolkit. The CPU implementation makes heavy use of the AVX2 features available in the Skylake architecture, and was compiled with gcc 7.3. The test matrices we consider are all of size $1\,000\,000 \times 1\,000\,000$ and diagonalizable with a unique largest magnitude eigenvalue, as this is required for the component-wise contraction property. The matrices are generated as band matrices with the aggregated number of nonzeros in a row on the main diagonal, and the values adjacent to the main diagonal set to -1. They differ in the number of nonzeros they carry in each row, the bandwidth, and the condition number.

Before focusing on the performance of CPMS on the HPC architectures, we visualize how the adaptive precision Jacobi interacts with the CPMS format[1] in Figure 6. In this example, the adaptive Jacobi uses a 2-segment CPMS format (left-hand side) and a 4-segment CPMS format (right-hand side) to solve an artificially created linear system of size $n = 10\,000$ containing 129 nonzeros in every row.[1] Initially, the iteration process only reads the heads. As the execution progresses, the significant accuracy is modulated on a per-component basis once the stagnation test indicates

[‡]http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/Configuration/Configuration_node.html

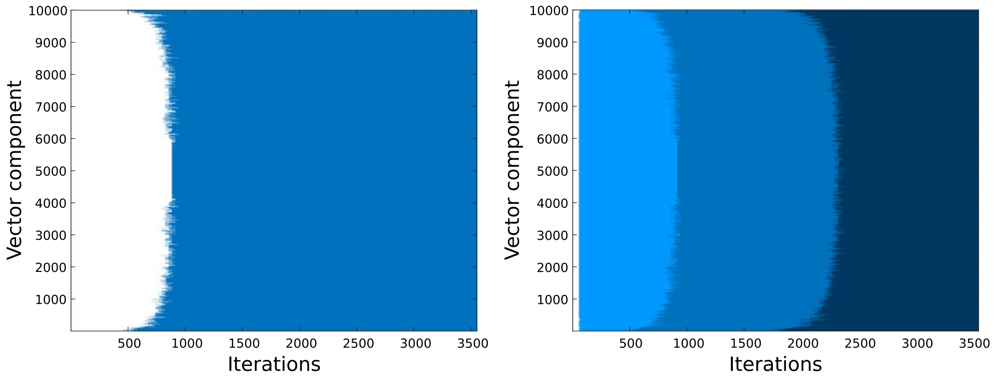


FIGURE 6 Accuracy needs in adaptive Jacobi in a 2-segment CPMS (left) and a 4-segment CPMS (right), respectively.[1] The white-colored area indicates only the head is accessed; the blue areas indicate additional significand segment reads

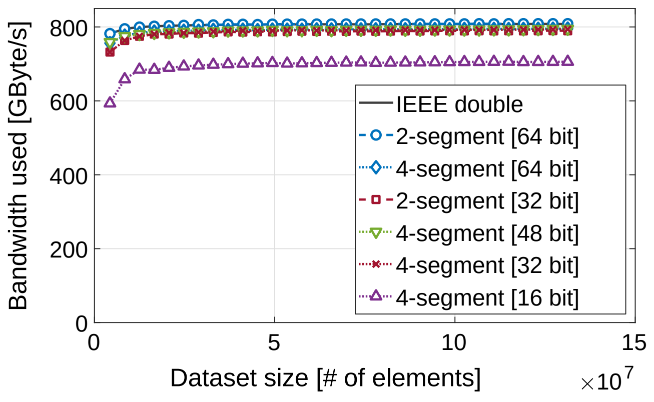


FIGURE 7 Bandwidth achieved by the distinct `CPMS` and the `fp64` memory access routines on the V100 GPU. The y-axis relates the bandwidth (GB/s) to the dataset size on the x-axis (# of floating point numbers transferred)

the need for higher accuracy. We note that, for the four-segment splitting, the initial 16-bit data access coming with only 4 mantissa bits fails to provide the accuracy needed to complete Jacobi iterations on this problem.

6.1 CPMS on GPUs

In this section, we focus on the GPU realization of the `CPMS` format. Similar to the results already reported for the NVIDIA P100 GPU,[1] we first assess the efficiency of the `CPMS` memory access routines and the overhead of reading the format specification carrying the information of how many significand segments are accessed. We note that the format specification information needs to be accessed only if the access accuracy is not known at compile time. Otherwise, format specification flags are not needed.

Figure 7 compares the bandwidth of the `CPMS` memory access routines with the bandwidth of the standard `fp64` access. The access routines for `CPMS` are not natively supported by hardware, and the hardware-specific implementations developed include the access to the segment information array, the element-wise decision of the segment access, some instruction logic to access the distinct segments in memory, the type cast to the double precision operating format, and the reassembling of the double precision format from head and significand segments. The results reveal that the `CPMS` memory access routines are highly competitive to the memory access routines natively supported by hardware. The `CPMS` routines accessing 64-bit accuracy are in par with the `fp64` access bandwidth, all exceeding 800 GByte/s. The 32-bit access and the 4-bit access of `CPMS` achieve only marginally lower bandwidth rates around 790 to 800 GByte/s. The notably lower bandwidth of the `CPMS` 16-bit access (700 GByte/s) can likely be linked to the hardware architecture being under-utilized.

More relevant than the bandwidth is the runtime needed for accessing values with various accuracy in the `CPMS` environment. Figure 8 shows the access cost of plain `CPMS` values (left-hand side) and `CPMS` values accompanied with the format specification information (right-hand side). As expected from Figure 7, the 64-bit access to `CPMS` is as fast as the access to `fp64`, independently of whether `CPMS` uses a 2-segment splitting or a 4-segment splitting (see left-hand side in Figure 8). 32-bit access to 2-segment `CPMS` is about twice as fast, and 16-bit access to 4-segment splitting takes about 28% of the time to access `fp64` values.

If the flag carrying the format specification is added, the data access cost is increased according to the memory volume overhead. For the 64-bit access to `CPMS`, the overhead is about 12%. For the 32-bit access, the overhead increases to 15% to 20%, for 4-segment and 2-segment `CPMS`, respectively. As elaborated, the 16-bit access does not fully saturate the memory bandwidth, and accessing the format specification information introduces about 25% overhead.

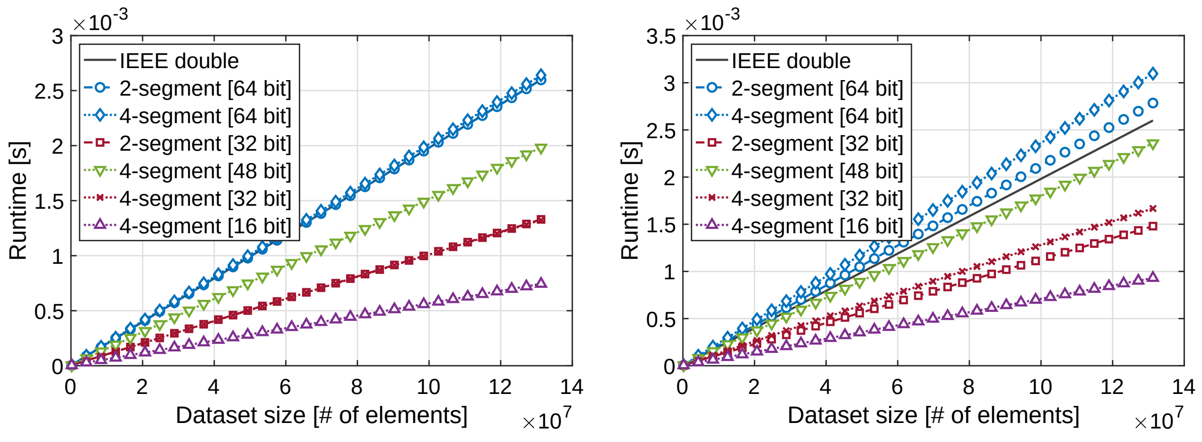
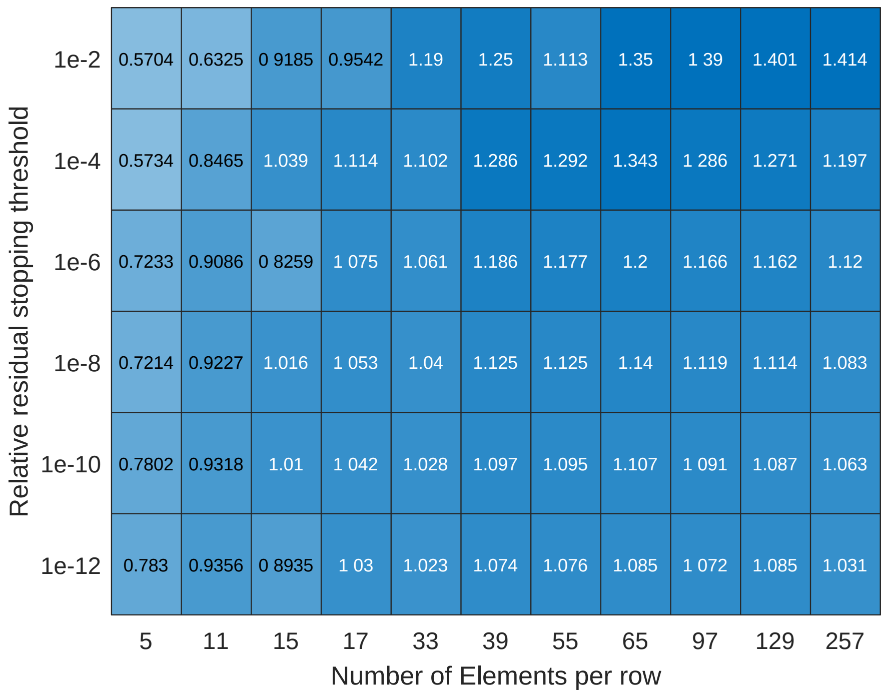


FIGURE 8 Runtime for reading and writing data in double precision or CPMS with the accuracy of the data accesses indicated in the brackets. Hardware architecture is the NVIDIA V100 GPU. The left-hand side plot shows the runtime for accessing the values only; the right-hand side plot includes the overhead of reading the significand-length information

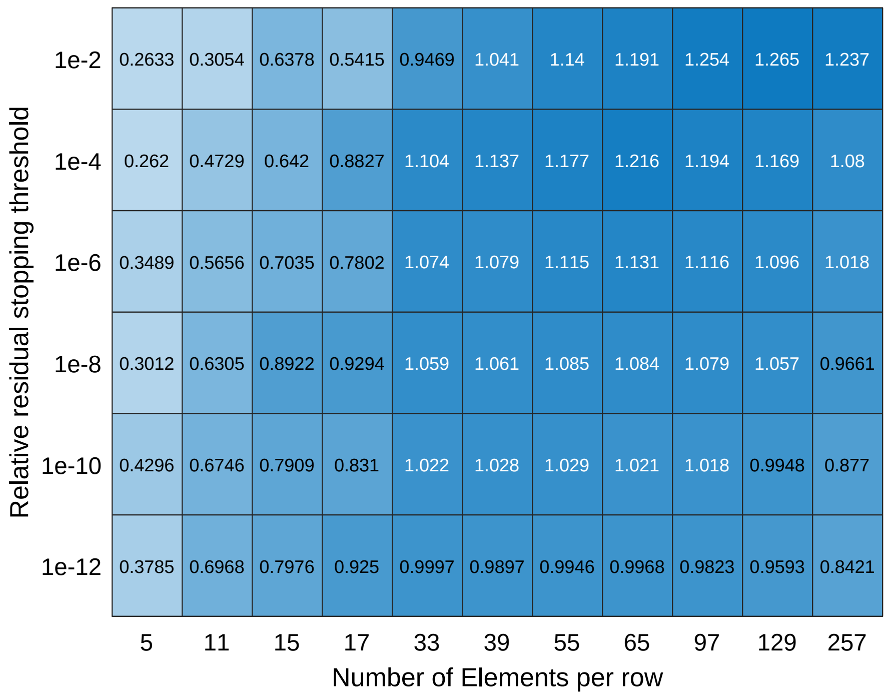


Relative residual stopping threshold	5	11	15	17	33	39	55	65	97	129	257
1e-2	0.5704	0.6325	0 9185	0.9542	1.19	1.25	1.113	1.35	1 39	1.401	1.414
1e-4	0.5734	0.8465	1.039	1.114	1.102	1.286	1.292	1.343	1 286	1.271	1.197
1e-6	0.7233	0.9086	0 8259	1 075	1.061	1.186	1.177	1.2	1.166	1.162	1.12
1e-8	0.7214	0.9227	1.016	1 053	1.04	1.125	1.125	1.14	1.119	1.114	1.083
1e-10	0.7802	0.9318	1.01	1 042	1.028	1.097	1.095	1.107	1 091	1.087	1.063
1e-12	0.783	0.9356	0 8935	1 03	1.023	1.074	1.076	1.085	1 072	1.085	1.031

FIGURE 9 Speedup factors of the adaptive precision Jacobi in a 2-segment CPMS realization on the V100 GPU

In the next experiment, we deploy the adaptive Jacobi inside the CPMS ecosystem. The adaptive Jacobi we use for this setting stores the system matrix in the SIMD-friendly ELL format.[13]

Previously presented experimental results[2] revealed that the adaptive Jacobi can exhibit some convergence delay compared to a plain Jacobi. One reason is that the mantissa extension detector, depending on the test periodicity ϕ, may not immediately identify stagnating components. Furthermore, the inevitably occurring rounding effects have to be reflected in the thresholds $\tilde{\delta}$ by accepting some variation in the convergence rate.[2] Both effects make it virtually impossible to precisely detect the mantissa extension point, which results in slight increase in the total iteration count. Finally, the stagnation detection itself incurs some runtime overhead. Complemented with CPMS, the question is whether the overhead accumulated from convergence delay, stagnation test, and slower access to 64-bit CPMS values can be compensated by the faster access to reduced precision values in some of the relaxation steps. To evaluate this interaction, we compare the time-to-solution of the adaptive precision Jacobi with a reference implementation of plain Jacobi in IEEE double precision. Both methods are run until they reach the same solution accuracy. We consider different relative residual stopping thresholds as Jacobi relaxations are often employed as smoother in multigrid methods or for providing rough solution approximations, eg, in approximate sparse triangular solves.[14] Taking the plain Jacobi as reference, Figure 9 reports the speedup factors of the adaptive precision Jacobi in a 2-segment CPMS realization for the distinct matrix/threshold combinations. The experimental results reveal that the adaptive precision Jacobi is particularly attractive (about 30% faster) for settings where a significant amount of matrix data has to be accessed in every iteration (many nonzero elements in every matrix row), and a large residual norm is acceptable (few



	5	11	15	17	33	39	55	65	97	129	257
1e-2	0.2633	0.3054	0.6378	0.5415	0.9469	1.041	1.14	1.191	1.254	1.265	1.237
1e-4	0.262	0.4729	0.642	0.8827	1.104	1.137	1.177	1.216	1.194	1.169	1.08
1e-6	0.3489	0.5656	0.7035	0.7802	1.074	1.079	1.115	1.131	1.116	1.096	1.018
1e-8	0.3012	0.6305	0.8922	0.9294	1.059	1.061	1.085	1.084	1.079	1.057	0.9661
1e-10	0.4296	0.6746	0.7909	0.831	1.022	1.028	1.029	1.021	1.018	0.9948	0.877
1e-12	0.3785	0.6968	0.7976	0.925	0.9997	0.9897	0.9946	0.9968	0.9823	0.9593	0.8421

FIGURE 10 Speedup factors of the adaptive precision Jacobi in a 4-segment `CPMS` realization on the V100 GPU

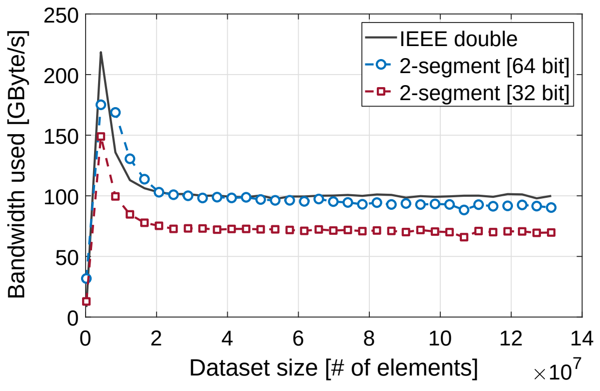


FIGURE 11 Bandwidth achieved by the distinct `CPMS` and the `fp64` memory access routines on the Intel Xeon Gold 6148 CPU

component iterations requiring the data with 64 bit accuracy). The faster access to the matrix values fails to compensate the overhead of the stagnation detection for problems with only few nonzeros in every row.

Figure 10 reports the same data for adaptive precision Jacobi in a 4-segment modular precision realization. A significant difference to the previously reported results on an NVIDIA P100 GPU[1] is that the 4-segment `CPMS` realization of adaptive precision Jacobi outperforms the Jacobi reference implementation for some problem settings. This is likely due to the more efficient 16-bit access on the newer Volta architecture, confirming that the segment configuration has to carefully align with the architecture characteristics.

6.2 CPMS on CPUs

As indicated by Figure 6, the benefits of using 16-bit accesses in the context of 4-segment `CPMS` are limited. Thus, the CPU experiments exclusively focus on the 2-segment `CPMS` format.

We again start the evaluation with an initial assessment of the memory bandwidth achieved by the `CPMS` memory access kernels. All kernels are parallelized using OpenMP (40 threads, one thread/core) and the data access is vectorized using AVX2 technology. In Figure 11 the bandwidth of `CPMS` memory access routines is compared to the standard `fp64` access. Ignoring the fluctuations in the size range where the dataset fits into cache, the `fp64` access achieves a sustained bandwidth of about 100 GByte/s. The 64-bit access of the 2-segment `CPMS` is highly competitive, achieving bandwidth of about 90 GByte/s. 32-bit `CPMS` access achieves about 70 GByte/s. Again, a more relevant metric is the data access time and the runtime overhead of accessing the format specification information. Figure 12 shows the access cost to the plain `CPMS` values (left-hand side) and the access cost to `CPMS` values accompanied by the format specification information (right-hand side). The results on the left-hand side of Figure 12 confirm the 64-bit access to `CPMS` is slightly slower than the `fp64` access. The 32-bit `CPMS` access is faster than the `fp64`

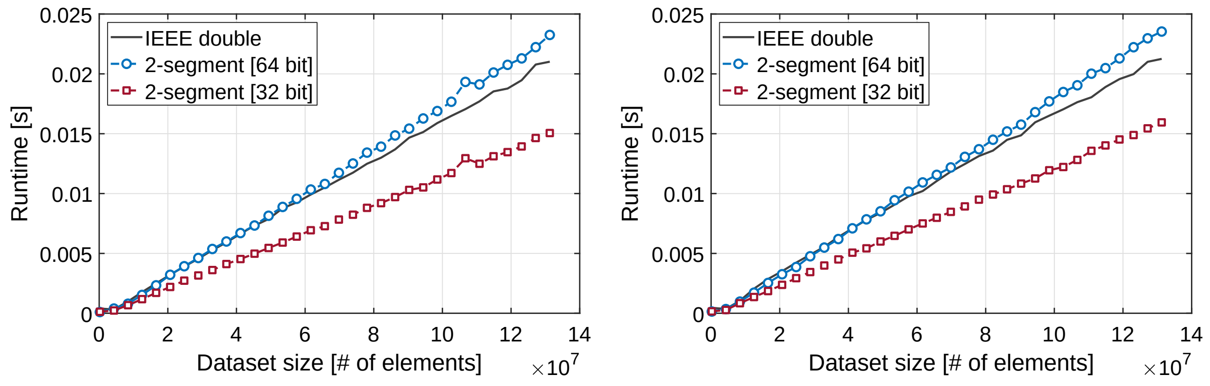
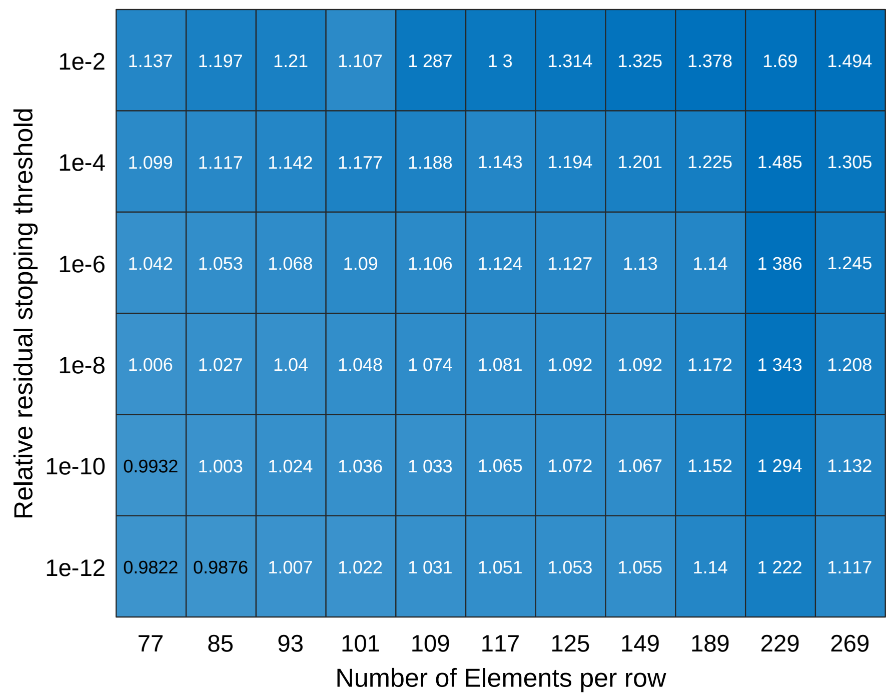


FIGURE 12 Runtime for reading and writing data in double precision or CPMS with the accuracy of the data accesses indicated in the brackets. The hardware architecture is the Intel Xeon Gold 6148 CPU. The y-axis relates the runtime (seconds) to the dataset size (# of floating point numbers transferred). On the left-hand side, only the data is accessed; the data visualized on the right-hand side includes the overhead introduced by also retrieving the format information



	77	85	93	101	109	117	125	149	189	229	269
1e-2	1.137	1.197	1.21	1.107	1 287	1 3	1.314	1.325	1.378	1.69	1.494
1e-4	1.099	1.117	1.142	1.177	1.188	1.143	1.194	1.201	1.225	1.485	1.305
1e-6	1.042	1.053	1.068	1.09	1.106	1.124	1.127	1.13	1.14	1 386	1.245
1e-8	1.006	1.027	1.04	1.048	1 074	1.081	1.092	1.092	1.172	1 343	1.208
1e-10	0.9932	1.003	1.024	1.036	1 033	1.065	1.072	1.067	1.152	1 294	1.132
1e-12	0.9822	0.9876	1.007	1.022	1 031	1.051	1.053	1.055	1.14	1 222	1.117

FIGURE 13 Speedup factors of the adaptive precision Jacobi in a 2-segment CPMS realization on the Intel Xeon Gold 6148 CPU

access. However, the benefits are smaller than for the GPU case, which is expected from the lower memory access rate reported in Figure 11. Furthermore, accessing the format specification information does not incur significant runtime overhead, as shown on the right-hand side of the figure.

Next, the adaptive precision Jacobi is embedded into the CPMS environment. The adaptive Jacobi we use for the CPU experiments stores the system matrix in CSR format.[15] Analogous to the GPU experiments, the plain Jacobi is taken as reference. Figure 13 reports the speedup factors of the adaptive precision Jacobi in a 2-segment CPMS realization for the distinct matrix/threshold combinations. As expected, the adaptive precision Jacobi outperforms the reference Jacobi, especially for settings where a significant amount of matrix data has to be accessed in every iteration (many nonzero elements in every matrix row), and a large residual norm is acceptable (few iterations requiring the data with 64 bit accuracy). For some problem configurations, ie, the test matrix containing 229 elements per row, we observe speedup values exceeding the expectations. These can likely be linked to effects related to a block of "head" segments efficiently matching the size of a cache line. As in the GPU case, the faster access to the matrix values fails to compensate the overhead of the stagnation detection for problems with only a few nonzeros in every row.

7 CONCLUDING REMARKS

We have presented a customized precision format that splits the significand of the baseline format into segments, ie, a strategy that preserves the exponent range of the floating point precision format. Complemented with the idea of radically decoupling the arithmetic format used in the arithmetic operations from the format used to store the data, the customized precision format enables much faster memory access if reduced accuracy in the memory operations is acceptable. Experimental results on server-line CPUs and GPUs revealed that realizing mixed precision algorithms in the customized precision format can render resource savings without impacting the memory footprint or the accuracy of the final result. We are convinced that the application field of customized precisions is much wider than what is presented in this work and envision the customized precision realization of Big Data analytics and Machine Learning applications.

ACKNOWLEDGMENTS

This work was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241, and by the CICYT project TIN2017-82972-R of the MINECO and FEDER. The authors want to acknowledge the access to the Piz Daint supercomputer at the Swiss National Supercomputing Centre granted under the project #d100 and the Juwels supercomputer at the Juelich Supercomputing Centre.

REFERENCES

1. Grützmacher T, Anzt H. A modular precision format for decoupling arithmetic format and storage format. In: *Euro-Par 2018: Parallel Processing Workshops Euro-Par 2018 International Workshops, Turin, Italy, August 27-28, 2018, Revised Selected Papers.* Cham, Switzerland: Springer; 2018:434-443.
2. Anzt H, Dongarra J, Quintana-Ortí ES. Adaptive precision solvers for sparse linear systems. In: Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing ACM; 2015; New York, NY.
3. Commitee IS. IEEE Standard for Modeling and Simulation (M&S) high level architecture (HLA)—framework and rules. IEEE Std. 1516-2000. 2000:i-22. https://doi.org/10.1109/IEEESTD.2000.92296
4. Horowitz M. 1.1 computing's energy problem (and what we can do about it). Paper presented at: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC); 2014; San Francisco, CA.
5. Saad Y. *Iterative Methods for Sparse Linear Systems.* 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2003.
6. Higham NJ. *Accuracy and Stability of Numerical Algorithms.* 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 2002. *Other Titles in Applied Mathematics*.
7. Anzt H, Heuveline V, Rocker B. Mixed precision iterative refinement methods for linear systems: convergence analysis based on Krylov subspace methods. In: Jonasson K, ed. *Applied Parallel and Scientific Computing: 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part II.* Heidelberg, Germany: Springer; 2010:237-247.
8. Carson E, Higham NJ. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J Sci Comput.* 2018;40(2):A817-A847. https://doi.org/10.1137/17M1140819
9. McCalpin JD. A survey of memory bandwidth and machine balance in current high performance computers. http://www.cs.virginia.edu/~mccalpin/papers/balance/
10. Anzt H, Dongarra J, Flegar G, Higham NJ, Quintana-Ortí ES. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurrency Computat Pract Exper.* 2019;31(6):e4460.
11. Mises RV, Pollaczek-Geiringer H. Praktische Verfahren der Gleichungsauflösung. *J Appl Math Mech.* 1929;9(2):152-164. https://doi.org/10.1002/zamm.19290090206
12. NVIDIA Corp. CUDA C Programming Guide. 9.0 ed. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
13. Bell N, Garland M. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report. NVR-2008-004. Santa Clara, CA: NVIDIA Corporation; 2008.
14. Anzt H, Chow E, Dongarra J. Iterative sparse triangular solves for preconditioning. In: Träff JL, Hunold S, Versaci F, eds. *Euro-Par 2015: Parallel Processing*. Vol. *9233.* Berlin, Germany: Springer; 2015:650-661. *Lecture Notes in Computer Science.*
15. Barrett R, Berry MW, Chan TF, et al. *Templates for the Solution of Linear Systems: Bu lding Blocks for Iterative Methods.* 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1994.