SIMD vectorization for simultaneous solution of locally varying linear systems with multiple right hand sides

Martin J. Kühn[1*], Johannes Holke[1], Annette Lutz[4], Jonas Thies[2], Melven Röhrig-Zöllner[1], Alexander Bleh[2], Jan Backhaus[2] and Achim Basermann[1]

[1*]Department for High-Performance Computing, German Aerospace Center (DLR), Institute for Software Technology, Linder Höhe, Cologne, Germany.
[2]Institute of Applied Mathematics, Delft University of Technology, Delft, the Netherlands.
[3]Department of Numerical Methods, German Aerospace Center (DLR), Institute of Propulsion Technology, Linder Höhe, Cologne, Germany.
[4]Department of Mathematics, Technische Universität Darmstadt, Darmstadt, Germany.

*Corresponding author(s). E-mail(s): Martin.Kuehn@DLR.de;

1 Introduction

For many applications, the simulation of turbomachinery requires the resolution of instationary flow phenomena to adequately predict, e.g., aeroelastic or aeroacustic behavior of a component. A good overview of modeling approaches for turbomachinery can be found in [1]. Assuming that the phenomena of interest are periodic in time, the equations can be expressed in the frequency domain using a Fourier series. One approach, called *harmonic balance* [2] is increasingly adopted in industrial turbomachinery design, as it leads to a reduction in computing times compared to the established Unstead Reynolds Averaged Navier-Stokes (URANS) method by up to two orders of magnitude [3].

In this paper we study the vectorization of this simulation method that exhibits an inherent level of vector-parallelism which results in the simultaneous solution of multiple sparse linear systems of equations which only differ by their main diagonal and right hand sides.

We implement and benchmark these techniques into the Sparse linear system solver library *Spliss*, a modern HPC library for solving block sparse linear algebra problems that is developed at the German Aerospace Center (DLR); see, e.g., [4]. Spliss is especially well-suited to solve the problem at hand, since its modular and templatized structure allows us to build onto its existing performant solvers.

We introduce a new data-type in Spliss, the *MultiScalar*, which contains a compile-time constant number of aligned scalars. Using this MultiScalar as the scalar type for the diagonal of a matrix and the right-hand side vector of a linear system we can apply Spliss' native linear system solvers to simultaneously solve the multiple equations in a single step. Since iterative solvers like GMRES or SoR repeatedly apply the system matrix to a vector many load operations from different memory levels may be necessary. With our approach, we only need to load the off-diagonal values of the matrix once instead of multiple times. In combination with suited SIMD vectorization of the required operations our method achieves a significant speed-up.

We compare two approaches to implement MultiScalars, one that relies on the compiler to insert the vector instructions and one which leverages the Vc [5] library. This is of particular interest as SIMD operations will hopefully be available with *std::simd* in a future C++ standard. We carry out thorough benchmark scenarios comparing the different approaches for various block-sizes and provide a recommendation on when to use which of the two approaches.

The paper is structured as follows. In Section 2, we first introduce the considered model problem. In Section 3, we give a brief introduction into current HPC architectures and present our particular implementations. In Section 4, we present numerical results considering vectorization ratio, timings, and Flops. Eventually, we draw a conclusion in Section 5.

2 Model problem

Computational fluid dynamics is an essential tool in the design of all types of fluid machinery. Of particular interest are the flows through turbomachinery, which convert fluid energy into rotation of a shaft and vice versa. This class of machines contains pumps, compressors, turbines, wind turbines, jet engines and gas turbines. The flow through these machines with all unsteady phenomena is fully described by the Navier-Stokes equations. However, numerically resolving all turbulent scales for the flows under consideration, direct numerical simulation (DNS), requires far too much computational effort to be affordable inside a design loop.

A common remedy is the temporal averaging of the equations and modeling the effect of all unsteady phenomena on the time-averaged flow. This approach is called Reynolds-averaged Navier-Stokes (RANS) and allows to exploit rotational symmetry and to reduce mesh resolutions to manageable magnitudes. The RANS equations have been the workhorse behind the design of most of today's turbomachinery. However their inherent negligence of the unsteady interaction of rotating and stationary turbomachinery components renders it infeasible for many types of aeroelasticity considerations.

This can be overcome by including larger unsteady flow phenomena into the computation by means of the unsteady RANS (URANS) method. While URANS captures unsteady interactions between components, it comes at the cost of no longer allowing to assume rotational symmetry of the flow field. Furthermore URANS computations exhibit a transient phase, where the initially assumed flow develops into a flow that satisfies the URANS equations. Having to simulate the full annulus until the flow converges to the final unsteady flow makes the URANS method about two orders of magnitude more expensive than RANS computations.

The fact that the interactions between rotating and stationary parts are periodic in time gives rise to methods that only model the time-periodic behavior. Consequently, an approach in between RANS and URANS is to solve the equations in the frequency domain and to restrict the unsteadiness to a selected number of base frequencies and harmonics thereof.

One approach for frequency-domain simulation in turbomachinery which allows for the nonlinear interaction between the mean flow and the harmonics is the harmonic balance approach [2].

After spatial discretization, e.g. through a finite-volume discretization, the semi-discrete Navier-Stokes equations take the form

$$\frac{\partial q}{\partial t} + R(q) = 0, \tag{1}$$

where R denotes the balance of fluxes and sources for the conservative flow state $q = (\rho, \rho u, \rho v, \rho w, \rho E)$ comprising density ρ, u, v, w, the momentum in the three spatial directions, and internal energy density E. The periodic behavior of this flow state can be approximated by taking the real part of a finite Fourier series of a base frequency ω and multiple non-negative harmonics $\hat{q}_k$

$$q(x,t) = \mathrm{Re}\left[\sum_{k=0}^{K} \hat{q}_k(x) e^{ik\omega}\right]. \tag{2}$$

The residual $R(q)$ can be transformed analogously. The time derivative transforms to a multiplication with $ik\omega$ in the frequency domain which yields a system of equations in the frequency domain

$$ik\omega\hat{q}_k + \widehat{R(q)}_k = 0. \tag{3}$$

This system can be solved by pseudo-time stepping, similar to the steady RANS problem, albeit in the frequency domain [6] with a pseudo-time τ, i.e.,

$$\frac{\partial \hat{q}}{\partial \tau} + ik\omega \hat{q}_k + \widehat{R(q)}_k = 0. \tag{4}$$

This set of equations is solved by an Euler backward approach for stability reasons. For the determination of the solution update $\Delta\hat{q} = \hat{q}^{m+1} - \hat{q}^m$, implicit pseudo-time stepping requires the solution of a linear system of equations

$$A\Delta\hat{q} = -\widehat{R(q)}. \tag{5}$$

Note, that due to the nonlinearity of R, the pseudo-time operator would couple all harmonics and therefore would contain K^2 coupled linear systems of equations. Each system is of size $(N_c \cdot N_d) \times (N_c \cdot N_d)$, where N_c denotes the number of grid cells and N_d the degrees of freedom per grid cell. In the case of finite-volume discretizations and when turbulence models are solved in a loosely coupled manner $N_d = 5$, where N_d is the number of physical variables describing the flow state. Larger values of N_d would arise in a Discontinuous Galerkin (DG) discretization. For details on DG, we refer to, e.g., [7, 8].

To avoid the quadratic growth of the linear system with K, the linearization may be based on the zeroth harmonic, yielding

$$\left(\left(\frac{1}{\Delta\tau} + ik\omega \right) I + \left. \frac{\partial R}{\partial q} \right|_{\hat{q}_0} \right) \Delta\hat{q}_k^m = -\widehat{R_k(q)}, \tag{6}$$

where I is the identity matrix.

This is a sequence of one real-valued and K complex-valued linear systems of size $(N_c \cdot N_d) \times (N_c \cdot N_d)$. In practical implementations the systems are represented as a sparse $N_c \times N_c$ matrix of dense blocks each being $N_d \times N_d$ in size. Note that only the entries on the main diagonal itself are complex-valued. However the whole dense $N_d \times N_d$ blocks on the diagonal are stored as complex. This is done for the ease of implementing block diagonal preconditioners such as occuring in Jacobi iterations or successive overrelaxation (SSOR).

These systems share their off-diagonal entries and differ only in the main diagonal and right hand sides

$$(D + J)\, \Delta\hat{q}_k^m = -\widehat{R_k(q)}, \tag{7}$$

where D designates the complex-valued diagonal matrix and J the linearization of the residual. This structure lends itself to a vectorized solution method, since the main diagonal, forming the majority of data, would only have to be transported once over the memory bus instead of $K+1$ times in the sequential approach.

In many CFD-related applications, we find typical block or stencil sizes of, e.g., 5 or 7 which do not lend themselves to SIMD operations naturally; see also the general introduction into computer architecture in the next section. The vectorization over multiple linear systems is thus an attractive alternative.

3 Computer architecture and SIMD vectorization

3.1 General introduction

Today's supercomputers feature multiple levels of parallelism; see, e.g., [9] for more details. On the highest level, multiple compute nodes communicate through a network. Each node consists of one or multiple CPU sockets, each with multiple cores. A node may also contain further accelerators such as GPUs or vector processors. Typically, the hardware in one node shares a memory address space even though there are multiple physical memories with different access speeds when accessed from different parts of the node (NUMA). A hierarchy of caches helps to bridge the gap between relatively slow main memory compared to the high performance of current processing units. On the lowest level, each CPU core (similar also for accelerators) consists of a pipeline of units that execute the desired instructions. One unit usually completes one instruction every cycle but needs multiple cycles to process it (latency). Similar to GPUs and vector processors, the CPU units allow to perform the same operation with multiple elements of data (Single Instruction Multiple Data: SIMD). Most Current CPU architectures have floating-point units with a SIMD width of 256 or 512 bit and allow to calculate one fused-multiply-add (FMA) instruction with vectors of 4 or 8 double-precision numbers respectively 8 or 16 single-precision numbers. CPUs for servers / HPC systems usually have 2 FMA units per core (superscalarity). To fully leverage the performance of one CPU core, one thus needs about hundred to several hundreds of independent floating-point operations to fill the pipeline.

There are some additional constraints for using SIMD instructions: ideally, the data should be stored in a contiguous array that starts at an aligned memory address (an address that is a multiple of the SIMD width). In addition, the code should access independent, consecutive chunks of data of size of the SIMD width. If the array length is not a multiple of the SIMD width, a remainder loop (without SIMD operations) is needed or a special masked SIMD operation must be added for the last few elements. Therefore, it is common practice to insert some zeros to obtain a data layout that allows better SIMD processing (padding) as the compiler can usually not adjust the data layout (which is used across multiple files and in external interfaces). It is then the task of the compiler optimization to rearrange the floating-point operations in such a way that they can be transformed into independent SIMD instructions. However, due to the inherent complexity of the required code transformations and due to possible pointer aliasing, compilers may not generate optimal SIMD instructions for a given algorithm. Therefore, SIMD-libraries like Vc [5] provide a suitable abstraction level to use SIMD features in a portable way across different CPU instruction sets.

benchmark	measurement
double precision performance (AVX2 FMA)	645 GFlop/s
memory bandwidth (AVX2 AXPY)	80 GByte/s

Table 1 Hardware characteristics of a 14-core Intel Xeon Scalable Processor Skylake Gold 6123 that is used for the numerical experiments measured using likwid-bench [11]. We use an axpy memory benchmark with an array size of 1 GB instead of the usual STREAM [10] benchmark as it better reflects the memory access pattern of our implementation. The hardware also supports AVX512 instructions (512 bit SIMD width instead of 256 bit with AVX2) but we do not use them in this paper.

This work focuses on the node-level performance. To ease the performance analysis, we consider the Roofline performance model [10] which states that the performance is either limited by the maximal in-core performance $P_{\max}$ or by data transfers. The maximal in-core performance depends on the mix of operations (and possible dependencies between them) and assumes that all required data is readily available in the nearest cache level. The data transfers are characterized through the main memory bandwidth b_m, respectively the bandwidth of the slowest data path used. Depending on the algorithm, the computational intensity I_c may change. The computational intensity specifies the number of (floating-point) operations per transferred byte. Combining these definitions, we obtain the Roofline performance:

$$P = \min(P_{\max}, I_c \cdot b_m).$$

If the data transfers are the limiting factor, the algorithm is called memory-bound. If, in contrast, the (floating-point) operations are the limiting factor, the algorithm is called compute-bound (or core-bound). Characteristic values for the peak memory bandwidth and the peak performance are shown in Table 1.

3.2 Realization in Sparse Linear System Solver library

The Sparse Linear System Solver library *Spliss* is a novel block sparse linear library that is developed for large scale CFD simulations; see [4]. Spliss is currently used in modern HPC CFD solver frameworks in aerospace and engineering, such as CODA [12] and TRACE [3]. Spliss is designed as a modern C++ library and relies heavily on templatization. This allows for a decoupling of abstract linear solvers and matrix format implementations on the one hand and concrete data types on the other hand.

Spliss employs distributed and shared memory parallelization. The former is realized via an internal abstraction layer which allows the usage of either two-sided MPI [13] or one-sided GASPI [14] communication as backend.

For shared memory parallelization Spliss uses the Alpaka (abstraction library for parallel kernels) framework [15], a performance portable, platform independent abstraction layer that allows using multiple (possibly different) accelerators concurrently. This enables Spliss to use, for example, OpenMP threads or CUDA for NVidia GPUs with the same high-level implementation.

Spliss implements a collection of common sparse linear system solvers such as CG and GMRES and allows for the application of preconditioners such as (Block)-Gauss-Seidel or SOR. It supports several sparse (block) matrix formats with either fixed or varying block size. Due to templatization these matrix formats and algorithms can operate on any arithmetic data type, such as *double*, *float*, *complex* or a user defined data type.

Realization of SIMD operations on multiple scalars

To efficiently execute SIMD operations on multiple scalars, we introduce the *MultiScalar* object as a custom data type which contains a compile-time constant number of aligned scalars. We have implemented two different realizations of MultiScalars, our naive implementation uses a member which is an array of scalars while the Vc-based implementation derives from *Vc::SimdArray*.

Together with the MultiScalar object itself, we need a mask object which is of the size of the MultiScalar and allows the comparison of MultiScalars.

The concept of the naive MultiScalar implementation is given in Fig. 1 (left). The Vc-based MultiScalar is only slightly more complex. In particular, it needs some additional lines of code for the definition of a corresponding *MaskType*. In Fig. 1 (right), we present our *VcAuto* implementation which lets Vc decide on the used vector length and which does not automatically introduce padding. For general padding, we would need to introduce another constant value *validEntries*. This number can then differ from *Size* which would be the size of the MultiScalar with padding. We present the realizations of simple MultiScalar comparison or addition operators in the appendix.

For the case of complex diagonals, we extend the concept of MultiScalar to *ComplexMultiScalar*. Since many operations on complex numbers (e.g., comparisons or additions) are based on a separate handling of real and imaginary parts, we use a *Complex* object which then holds two MultiScalars; one for the real and one for the imaginary part, each purple box representing one scalar value; see Fig. 2.

Mixed scalar type matrices

Owing to the application of harmonic balance problems, where, e.g., the diagonal blocks of a matrix can be complex-valued while the off-diagonal blocks only contain real-valued entries, Spliss offers the so called *CompositeMatrix*. For such a matrix A, we define A_C as the set of (diagonal) matrix blocks that are complex-valued and A_R as the set of (off-diagonal) blocks that are real-valued. Then $Ax = A_C x + A_R x$ and the matrix is split up into multiple separate parts, each comprising a homogeneous datatype. This allows for a very flexible and memory efficient assembly of mixed data type matrices, or even matrices featuring additional matrix-free operators. For the sake of simplicity, we assume in our description that A_C and A_R are stored in a meaningful way such that the above matrix-vector-products can be executed.

```
// Naive MultiScalar
template<typename T, size_t N>
 class MultiScalar{
public:
  // Mask for MultiScalar
  using MaskType = MultiScalar<bool,
        N>;

  // constructors, operators, ...

protected:
  // Array of values in MultiScalar
  T values_[N];
};
```

```
// Vc-based MultiScalar
template<typename T, size_t N>
 class MultiScalar: public
       Vc::SimdArray<T, N> {
public:
  using BaseClass =
        Vc::SimdArray<ScalarType, Size>;
  using VcMaskType = typename
        BaseClass::mask_type;

  // MaskType inheriting from
        VcMaskType
  struct MaskType: public VcMaskType {
    using VcMaskType::VcMaskType;
    MaskType(const VcMaskType&
          other) : VcMaskType(other){}

    // Performs the logical and for two
          MaskTypes.
    MaskType operator&&(const
          MaskType &other) const {
      MaskType tmp(*this);
      for ( size_t i = 0; i < tmp.size();
            i++)
        tmp[i] = tmp[i] && other[i];
      return tmp;
    }

    // operator|| similar to operator&&
  };

  // constructors, operators, ...
 };
```

Fig. 1 Naive (left) and Vc-based (right) MultiScalar implementations.

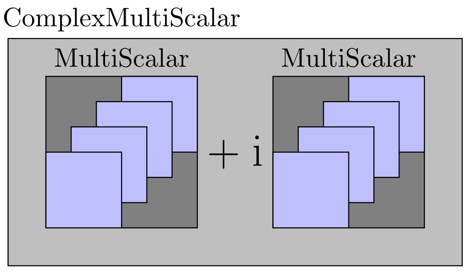

Fig. 2 *ComplexMultiScalar* composed out of two *MultiScalar*s, one for the real- and one for the complex-valued entries.

A very simple CompositeMatrix with only one diagonal (in purple) that can have a different data type than the green off-diagonal blocks is shown in Fig. 3 (left). The more relevant use case with multiple diagonals (i.e., MultiScalars on the diagonal) that have a different data type than the off-diagonal blocks can be found in Fig. 3 (right).

The downside of the current implementation is that the vectors have to be loaded from memory once for the diagonal and once for the off-diagonal part. Nevertheless, due to the focus on block matrices, this effect becomes

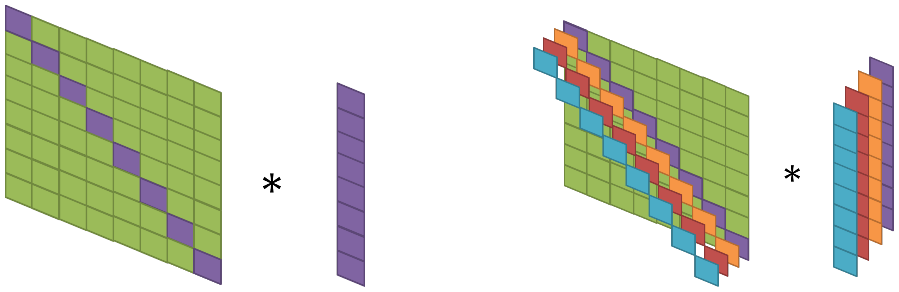

Fig. 3 Matrix-vector-multiplication of a *CompositeMatrix* with a single diagonal (left) and a MultiScalar with four entries on the diagonal (right).

smaller with larger block sizes, since the matrix (which is loaded once) requires $\mathcal{O}(N_c N_d^2)$ data transfers whereas the vector only requires $\mathcal{O}(N_c N_d)$ transfers.

In this work we use the CompositeMatrix to define (Complex)MultiScalar entries on the diagonal and real-valued entries on the off-diagonal but other choices are also possible. The application of real-valued off-diagonal blocks on either complex- or real-valued right hand sides then saves memory transfers and operations since it avoids storing (and calculating with) zeros for the imaginary part of the matrix entries.

4 Numerical results

We consider a variety of different systems to test the performance of our implementations. All test cases are based on CompositeMatrices with a maximum of seven nonzero blocks per row, i.e., up to six off-diagonal blocks (three left and three right of the diagonal with a distance of 1, 10, and 100 blocks). All these blocks (diagonal and off-diagonal) are fully dense and the whole system matrix is (block) sparse. Off-diagonal blocks are always real-valued. For the diagonal blocks, we consider either real or complex entries. We consider examples with single diagonals as well as with MultiScalar or ComplexMultiScalar diagonals of different size.

We consider two edge cases with either small 5×5 or large 240×240 blocks. In case of small blocks, we consider a matrix with $5 * 1.5$ million rows. This leads to 37.5 million nonzeros for the diagonal blocks and 225 million nonzeros in the off-diagonal blocks of the matrix. For the larger block size, we consider a matrix with $240 * 700$ rows leading to 40.3 million nonzeros for the diagonal blocks and 229 million nonzeros in the off-diagonal blocks.

The size of the matrices in the memory of the different tests differs by the data type of the diagonal as well as the precision used. We consider test cases with single precision (denoted FP32) or double precision (denoted FP64).

Let us provide an example on the data size for the case of 5×5 blocks, four complex diagonals and double precision. Using a ComplexMultiScalar of size 4, the required storage for the diagonal blocks is then given by

$$S_D = N_{NZ} * N_{bd} * N_{MS} * N_{RC} = 37.5 * 10^6 * 8 * 4 * 2 = 2.4 \text{ [GB]},$$

where N_{NZ} denotes the number of nonzero entries stored in the diagonal, N_{bd} is the size in bytes of one double, N_{MS} specifies the number of entries in the Multiscalar and N_{RC} is a multiplier to distinguish between real- and complex-valued entries. Accordingly the off-diagonal needs

$$S_{OD} = 225 * 10^6 * 8 * 1 * 1 = 1.8 \text{ [GB]}$$

bytes. Four complex double-precision vectors result in

$$S_V = 5 * 1.5 * 10^6 * 8 * 4 * 2 = 0.48 \text{ [GB]}$$

We use *gcc* 10.2 and *Vc* 1.4.1. We perform numerical tests on a single socket with the hardware characteristics shown in Table 1. We do not use AVX512 SIMD instructions as AVX-512 will only be available with Vc 2[1]. In addition, the CPU reduces the frequency when AVX512 instructions are executed. This makes AVX512 instructions not always beneficial and complicates the performance analysis.
We divide the numerical results section into four different sections:

Bandwidth saturation: In Section 4.1, we briefly discuss the compute intensity of our application and show that we obtain saturating behavior.
Vectorization behavior: In Section 4.2, we show the vectorization behavior of the different implementations.
Timings and Flops/s: In Section 4.3, we consider timings and Flops per second for different numbers of diagonals and benchmarks for the different implementations.
Realistic Example: In Section 4.4, we validate the findings using a linear matrix obtained from the CFD solver TRACE within a harmonic balance context.

4.1 Bandwidth saturation

For the matrix with 5×5 blocks and $1.5 \cdot 5$ million rows in complex arithmetic, we need

$$37.5 * 10^6 * 4 * 2 * 4 = 1.2 \text{ [GFlop]}$$

for the multiplication of 4 complex block diagonals with 5×5 blocks with 4 complex vectors. For the off-diagonal part, we have one matrix with 225 million real entries, resulting in

$$225 * 10^6 * 2 * 2 * 4 = 3.6 \text{ [GFlop]}$$

[1] https://github.com/VcDevel/Vc

Therefore, we obtain the compute intensities

$$I_{C,\text{diag}} = \frac{1.2}{2.4 + 3 * 0.48} \approx 0.3 \text{ [Flop/Byte]}$$
$$I_{C,\text{off-diag}} = \frac{3.6}{1.8 + 3 * 0.48} \approx 1.1 \text{ [Flop/Byte]}$$

for the block-diagonal part and the off-diagonal parts of the computation. For bigger blocks and more vectors, the compute intensity increases (e.g. to ~ 5.7 [Flop/Byte] for the off-diagonal part of the matrix with 240×240 blocks and 8 vectors). From Table 1, the machine intensity is

$$I_M = \frac{645}{80} \approx 8 \text{ [Flop/Byte]}.$$

Thus, for the Roofline performance model all variants are *memory-bound*. However, for the cases with multiple diagonals the compute intensity is close enough to the machine balance that the SIMD vectorization affects the performance. This is the regime (compute intensity close to the machine intensity) where the Roofline model is too optimistic in the sense that it assumes a perfect overlap of data transfers and computations; more sophisticated performance models like the Execution-Cache-Memory (ECM) model[16] could predict the performance more accurately but we will focus on the generic implementation and SIMD vectorization here.

In the following, we show that our model problems still feature bandwidth-saturating behavior. We compute the MatVec benchmark for the model problems with 5x5 and 240x240 blocks on the diagonal. The MatVec benchmark conducts one matrix-vector-product with both the block-diagonal and the off-diagonal parts. We run this with MultiScalars of either 1, 4, or 8 scalars on these diagonals on 1 to 14 cores; cf. Fig. 4. All benchmarks are executed on the machine depicted in Table 1. We compare the bandwidth to the bandwidth obtained with the *AXPY* benchmark in LIKWID[11]. We chose the *AXPY* benchmark as reference as it has a similar load-to-store ratio. All computations update the vector (no nontemporal stores).

We can see from Figure 4 that all cases with *VcAuto* achieve a high fraction ($> 75\%$) of the peak bandwidth and that they feature saturating behavior. Nevertheless, most cases are not completely saturated (more cores could further increase the performance), especially the case with 240×240 blocks and 1 diagonal scales almost linearly with the number of cores. As the cases with 240×240 blocks and more diagonals achieve a higher bandwidth even though they have a higher compute intensity, this indicates sub-optimal compiler optimization. The less degressive scaling of the implementation without Vc in comparison already indicates a better utilization of the compute performance when using Vc. This observation will be further investigated in the following sections.

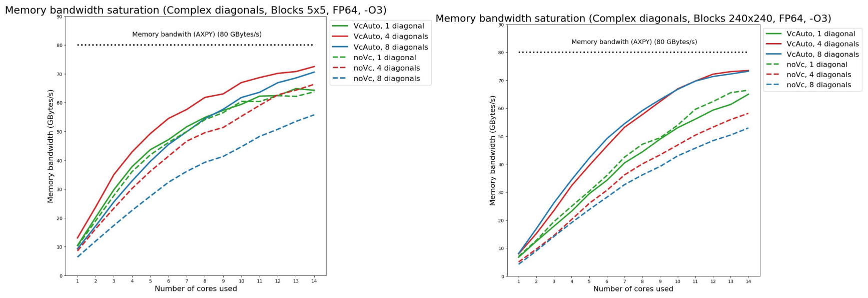

Fig. 4 Memory bandwidth of the MatVec-benchmark measured with LIKWID for 1, 4, and 8 diagonals with dense **5x5** or **240x240** blocks of **complex** entries in **double precision**. *VcAuto* uses the Vc-based MultiScalar where Vc decides on the used vector lengths. *noVc* also solves all systems simultaneously but only uses a naive array-based MultiScalar implementation.

4.2 Vectorization behavior

In this section, we compare the naive Multiscalar implementation, denoted *noVc*, with the padded implementation, denoted *VcPad*, as well as with the implementation, where potential padding is decided by Vc, denoted *VcAuto*. Finally, we also provide compiler-obtained vectorization for a sequential solution of the systems with different diagonals. Note that padding of, e.g., a MultiScalar of size 3 to size 4 can help to make the code more suitable for SIMD vectorization. However, it comes at the cost of storing and transferring additional zeros (4/3 of the original date transfers required for the vectors).

As expected, we see from Fig. 5 that vectorization of VcPad is always at 256 bit since MultiScalars are padded to this size. For VcAuto, we observe that practically no padding is used. For instance, in all cases, we see that vectorization for VcAuto MultiScalar of size 7 is done with one third to lengths 64, 128, and 256 bits. For Vc-based implementations, we see that vectorization does not depend on optimization flags (see left column for *O2* and right column for *O3* in Fig. 5). We explicitly consider both *O2* and *O3* optimization, as it is common for scientific and engineering codes, to enable only the *O2* optimization level.

For sequential solutions as well as the naive noVc implementation, we do not see any vectorization by the compiler with *O2* flag; see left column in Fig. 5. We see modest vectorization with *O3* flag but this is far below the vectorization as achieved by Vc-based implementations; see right column in Fig. 5.

4.3 Timings and Flops

The results on timings and Flops can be divided into two different categories. In Fig. 6 and Fig. 7, we consider the MatVec benchmark for one up to 16 diagonals and 5x5 or 240x240 sized blocks, respectively. Results are presented for *O2* and *O3* optimization, single and double precision as well as real- and complex-valued diagonals. In Fig. 8-Fig. 11, we consider five different benchmarks for

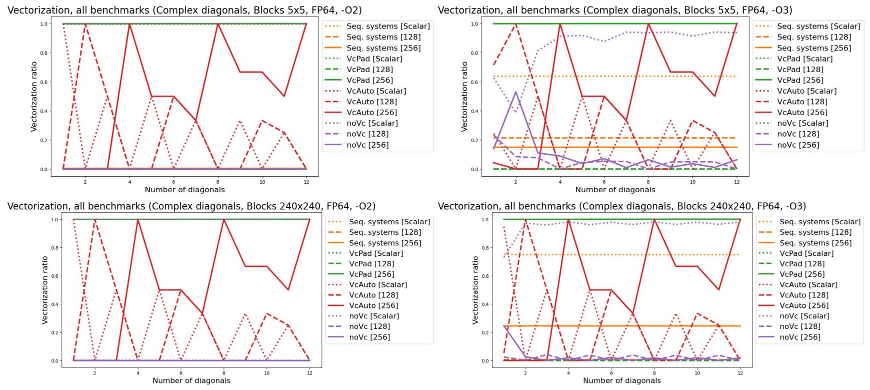

Fig. 5 Vectorization ratios for *-O2* (left column) and *-O3* optimization (right column) for 5x5 (top row) and 240x240 (bottom row) blocks of complex entries on the diagonal. *Seq. systems* corresponds to a sequential solve of the systems. *VcPad* uses a padded Vc-based MultiScalar on the diagonals to solve all systems simultaneously. Similarly, *VcAuto* uses a Vc-based MultiScalar where Vc decides on the used vector lengths. *noVc* also solves all systems simultaneously but only uses a naive array-based MultiScalar implementation. Dotted lines correspond to scalar execution, dashed lines represent the share of FP_128 vectorization and solid lines FP_256 vectorization. FP_512 is not used.

different block sizes and different precision. We consider *O3* optimization and real- as well as complex-valued diagonals.

In Fig. 6 and Fig. 7, we see that different optimization flags make a huge difference for the naive MultiScalar implementation while for single systems as well as Vc-based implementations the optimization gain is smaller. For Vc, *O2* optimization already yields best performance in some cases.

For Vc-based implementations, we see that the best performance is obtained for the number of diagonals that fits a multiple 256-byte width, i.e., eight or 16 for single precision and four or eight (in some cases also 12 or 16) for double precision.

We generally see that the padded Vc-MultiScalar behaves suboptimally for systems with a small number of diagonals. This is due to the relatively large padding to four (double precision) or eight (single precision) diagonals. Here, the Vc-MultiScalar, where vector widths are derived automatically (*VcAuto*), yields much better results. However, for a badly chosen number of diagonals (i.e., seven for double precision), *VcAuto* conducts three SIMD operation of length 256, 128, and 64 bytes instead of two 256-bytes operations for VcPad; see also Fig. 5).

For a small number of diagonals, *VcAuto* and the naive *noVc* approach perform similarly well. In case of small block sizes (5x5), the *noVc* performs best for double precision and more than four diagonals (except eight). On the other hand, for eight double precision diagonals *VcAuto* still performs best on small block sizes (5x5). For single precision, independent of the block size, or double precision and large block sizes (240x240) the picture is clear. Here, *VcAuto* performs best for all multiples of 256-bytes widths.

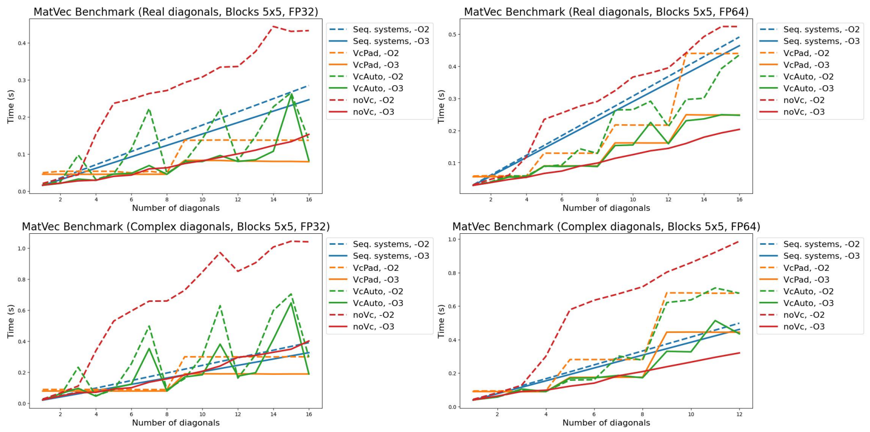

Fig. 6 MatVec Benchmark timings for diagonals with dense 5x5 blocks of real (top row) and complex (bottom row) entries and single (left) and double (right) precision. Dashed lines represent execution with *-O2* optimization, solid lines represent execution with *-O3* optimization. Other notation as in Fig. 5.

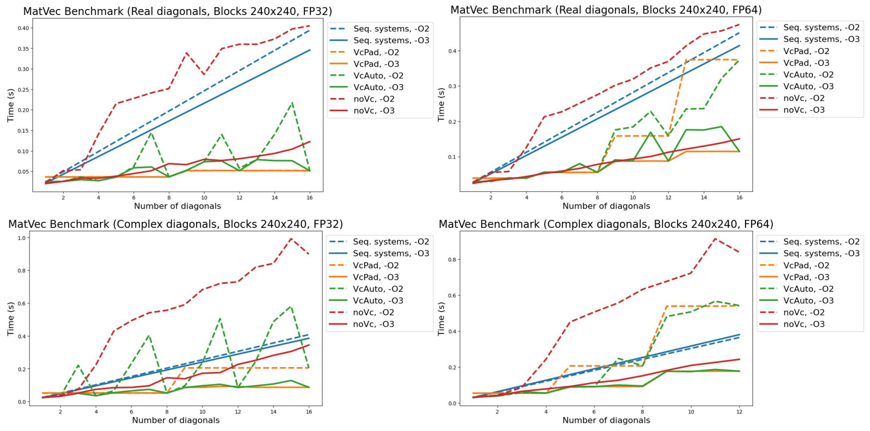

Fig. 7 MatVec Benchmark timings for diagonals with dense 240x240 blocks of real (top row) and complex (bottom row) entries and single (left) and double (right) precision. Dashed lines represent execution with *-O2* optimization, solid lines represent execution with *-O3* optimization. Other notation as in Fig. 5.

In Fig. 8-Fig. 11, we consider five different benchmarks. Besides the previous MatVec benchmark, we test a colored matrix vector product *ColMatVec* with three different colors. Furthermore, we test a block Jacobi *BlockJac* and block Gauss-Seidel *BlockGS* iteration scheme, where the preconditioner is computed as the LU decomposition of the block diagonal matrix. In the resulting timings and Flops, we only consider the iteration scheme, not the set up of the preconditioner. Finally, we also test a *GMRES* iteration scheme. For all three iterative schemes, a modest number of iterations smaller 10 is conducted.

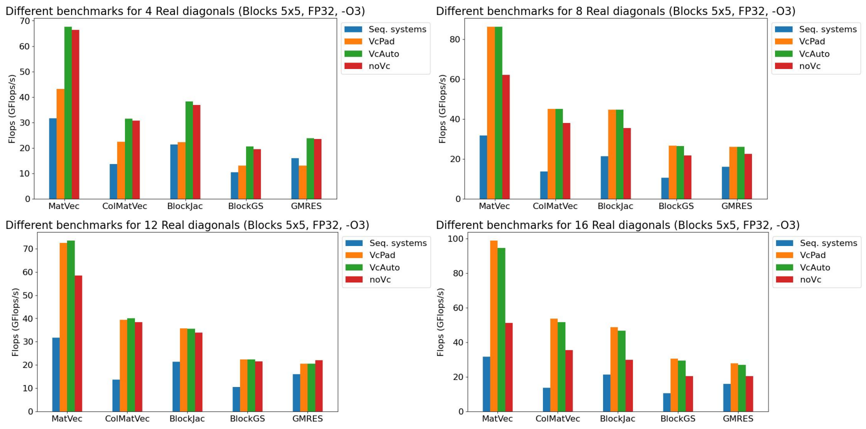

Fig. 8 Obtained performance with different benchmarks for 4, 8, 12, and 16 diagonals with dense **5x5** blocks of **real** entries in **single precision**. Other notation as in Fig. 5.

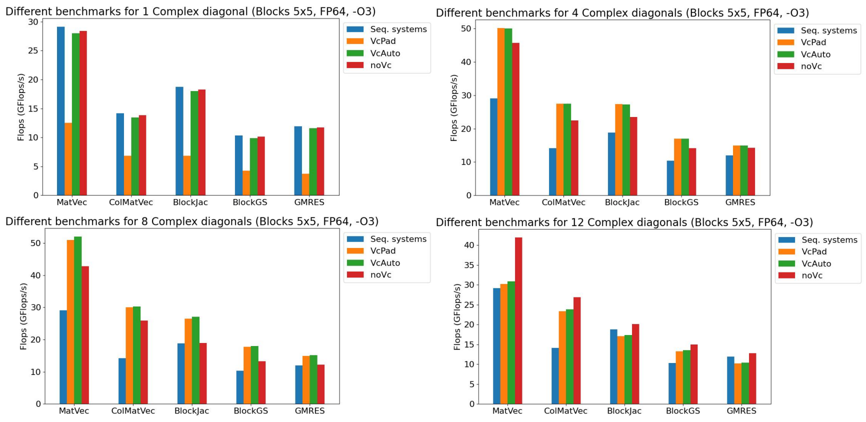

Fig. 9 Obtained performance with different benchmarks for 1, 4, 8, and 12 diagonals with dense **5x5** blocks of **complex** entries in **double precision**. Other notation as in Fig. 5.

As we do not want to compare the different benchmarks (e.g., BlockJac vs. GMRES), the number of iterations is not important.

The Flop count that we present only includes intended Flops that we need for the corresponding result. That means, that an addition of two MultiScalars with one value each padded up to four values will only result in one Flop, not in four. Consequently, the GFlops/s obtained with *VcPad* are low for small numbers of diagonals.

Except for the case of 12 complex diagonals in double precision and small 5x5 blocks, the *VcAuto* MultiScalar implementation always achieve the most Flops per second (or are within a range of some percent of the best performance).

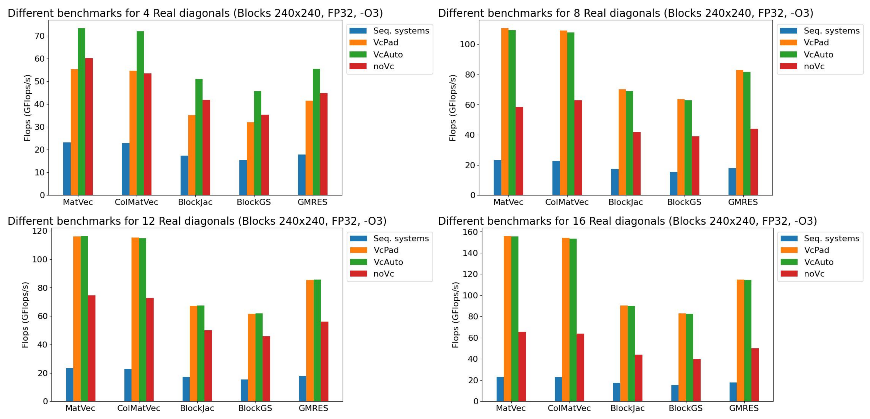

Fig. 10 Obtained performance with different benchmarks for 4, 8, 12, and 16 diagonals with dense **240x240** blocks of **real** entries in **single precision**. Other notation as in Fig. 5.

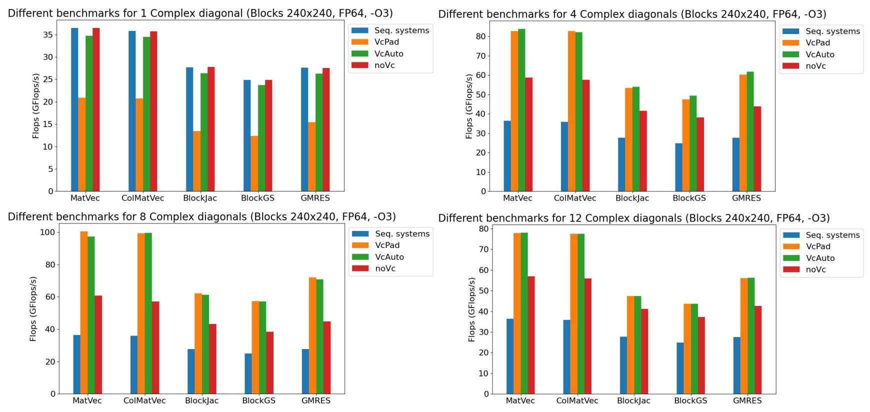

Fig. 11 Obtained performance with different benchmarks for 1, 4, 8, and 12 diagonals with dense **240x240** blocks of **complex** entries in **double precision**. Other notation as in Fig. 5.

4.4 Realistic example from the CFD solver TRACE

To validate the benchmark results, we tested the MultiScalar implementation in a realistic use case. Fig. 12 shows the sparse matrix structure extracted from the CFD solver TRACE [3] for the simulation of a transsonic compressor fan. The discretization results from a structured mesh consisting of about one million finite volume cells. Each entry of the sparse matrix consists of a dense 5x5 matrix block. A harmonic balance solution which features N higher harmonics will require N additional equally structured linear matrices with a complex diagonal to be solved. As before, we solve these systems using different approaches. First, we use the classical approach without MultiScalars and solve the systems sequentially. Second and third, we compare the performance of our own naive MultiScalar implementation against one implementation using the Vc library. In TRACE, typically, a colored block Gauss-Seidel approach is

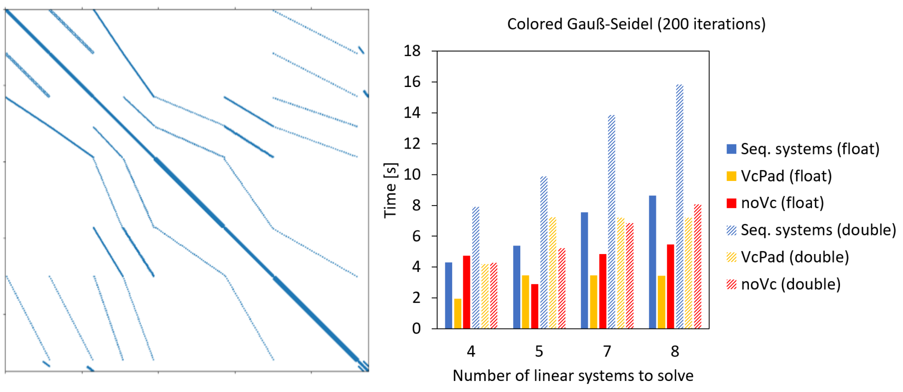

Fig. 12 Left: The structure of the linear system Matrix. Right: Results for four different numbers of higher harmonics and double/single precision.

used with a fixed number of iterations. The library Spliss inverts the 5x5 blocks on the diagonal using an LU decomposition. To exclude effects of distributed MPI parallelization, the system is solved on a single node comprising 32 cores (System specification: Sky Lake Xeon(R) Silver 4216). The results are shown in Fig. 12 for single precision, double precision and different numbers of higher harmonics considered, i.e., $N \in \{4, 5, 7, 8\}$. To allow for comparison, we used a fixed number of 200 iterations.

Let us note that all systems represented by one MultiScalar are always computed together. The iteration process can then be stopped if either one or all approximations have reached the convergence criterion. This means that, under certain conditions, some systems are solved unnecessarily precise. Such kind of algorithmic losses are not considered here.

For the case of sequentially solved systems a proportional increase in computing time can be observed as expected. For double precision the computing time is about twice the amount as for single precision. For either four or eight entries of a MultiScalar, the Vc-based implementation yields an improvement of about a factor of two. For odd numbers which do not fully fill a SIMD register, the own (naive) implementation relying on the compiler is faster in some cases. For numbers filling a SIMD register or being a multiple of it, the naive implementation is slower. Compared to sequential solution of the systems, the use of Vc-based implementation of the MultiScalar always reduces the computational time significantly.

5 Conclusion

In this paper we presented three implementations of MultiScalars. The first implementation is a naive C++ implementation while the second and third one make use of the Vc library [5]. Hereby, we conduct the parallel solution of linear systems, which only differ for a limited number of matrix entries. These systems may naturally arise from computational fluid dynamics problems as described in Section 2. The parallel solution of these systems is conducted using SIMD operations allowing the concurrent processing of, e.g., four double precision numbers.

Since our model problems are memory-bound, we see that we benefit from lower memory transfer using our CompositeMatrices. However, we also benefit from SIMD parallelism realized on the diagonal blocks of the matrices. We observe limits on default compiler vectorization for naive MultiScalar implementations (denoted *noVc*); on the other hand, we see that Vc-based implementations (denoted *VcPad* and *VcAuto*) vectorize well; see Fig. 5. Insights into the different implementations is given in Fig. 1. However, our Vc-based implementations (*VcPad* and *VcAuto*) do not perform best for all use cases. As vectorization is well achieved with these implementations, the number of systems to be solved in parallel should be chosen carefully. For instance, for 256 bit sized SIMD registers and double precision systems, our padded Vc-implementation always solves multiples of four systems. This means that for just two systems to be solved, two systems are solved in padding. The *VcAuto* implementation would then only vectorize with length 128 bit and solve both systems without overhead. However, this implementation is disadvantageous if seven double precision systems need to be solved. Then vectorization is done in 256, 128, and 64 bit size; see, e.g., Fig. 5.

Chosing the number of systems to be solved in parallel in accordance with the SIMD width can lead to a substantial reduction of computation time. As expected, the solution of systems in parallel is substantially faster than sequentially solving these systems. Additionally, the Vc-based implementations also outperform the naive MultiScalar implementation considerably; see Figures 6-Fig. 11. This effect grows with the block sizes.

For a realistic, memory-bound example with small-sized 5x5 diagonal blocks, we finally achieve a speedup of factor two compared to a sequential solution of four or eight systems. We also obtain a significant reduction in computation time by using the Vc-based MultiScalar implementations against the presented naive implementation. Further advantages of SIMD execution could result from lower total energy consumption. However, this was not measured and is subject to future research.

Acknowledgments. The authors gratefully acknowledge the funding of the DLR project TOSCANA, which is part of the German government's aviation research program under grant agreement number FKZ20X1704A. The authors also thank Jens Jägersküpper, Olaf Krzikalla, Jasmin Mohnke, Arne Rempke, Daniel Vollmer and others who contributed to Spliss.

Conflict of interest

The authors declare that they have no conflict of interest.

Data availability

No particular data sets were used to conduct this study. The nonzero patterns of the test cases can be constructed easily, the values of the matrices are arbitrary as time to convergence of the iterative solvers was not the focus of the study.

Authors' contributions

All authors contributed to the study conception and design. Coding and analysis were performed by Jonas Thies, Johannes Holke, Martin J. Kühn, Annette Lutz, Melven Röhrig-Zöllner, Alexander Bleh and Jan Backhaus. The first draft of the manuscript was written by Martin J. Kühn, Johannes Holke, Melven Röhrig-Zöllner, Alexander Bleh and Jan Backhaus and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Further declarations

- Ethics approval: Not applicable
- Consent to participate: Not applicable
- Consent for publication: Not applicable
- Code availability: Code snippets are shared within this article. The full library of Spliss is not publicly available.

Simple comparison or addition operators for the *naive* MultiScalar.

```
/// Computes the component-wise sum of this MultiScalar and another
    MultiScalar.
template<typename InputScalarType>
MultiScalar operator+(const MultiScalar<InputScalarType, Size>&
    other) const
{
  return MultiScalar(*this) += other;
}

/// Computes the component-wise sum of this MultiScalar and a scalar.
template<typename InputScalarType,
    std::enable_if_t<std::is_arithmetic<InputScalarType>::value,
    bool> = true>
MultiScalar operator+(const InputScalarType other) const
{
  return MultiScalar(*this) += other;
}

/// Tests this MultiScalars component-wise for equality against
    another MultiScalar.
template<typename InputScalarType>
```

```
MaskType operator==(const MultiScalar<InputScalarType, Size>& other)
    const
{
  MaskType result;

  for (size_t i = 0; i < Size; i++)
  {
    result[i] = (*this)[i] == other[i];
  }
  return result;
}

/// Tests this MultiScalar component-wise for equality against a
    scalar.
template<typename InputScalarType,
    std::enable_if_t<std::is_arithmetic<InputScalarType>::value,
    bool> = true>
MaskType operator==(const InputScalarType& other) const
{
  MaskType result;

  for (size_t i = 0; i < Size; i++)
  {
    result[i] = (*this)[i] == other;
  }
  return result;
}
```

Simple comparison or addition operators for the *Vc-based* MultiScalar.

```
// Computes the component-wise sum of this MultiScalar and another
    MultiScalar.
template<typename InputScalarType>
MultiScalar operator+(const MultiScalar<InputScalarType, Size>&
    other) const
{
  MultiScalar<ScalarType, Size> res;
  res = *static_cast<ConstBaseClass*>(this) +
      static_cast<ConstBaseClass&>(other);
  return res;
}

// Computes the component-wise sum of this MultiScalar and a scalar.
template<typename InputScalarType,
    std::enable_if_t<std::is_arithmetic<InputScalarType>::value,
    bool> = true>
MultiScalar operator+(const InputScalarType& other) const
{
  return *static_cast<ConstBaseClass*>(this) + ConstBaseClass(other);
}

// Tests this MultiScalars component-wise for equality against
    another MultiScalar.
template<typename InputScalarType>
MaskType operator==(const MultiScalar<InputScalarType, Size>& other)
    const
{
  return MaskType(*static_cast<ConstBaseClass*>(this) ==
      static_cast<ConstBaseClass&>(other));
}

// Tests this MultiScalar component-wise for equality against a
    scalar.
template<typename InputScalarType,
    std::enable_if_t<std::is_arithmetic<InputScalarType>::value,
    bool> = true>
MaskType operator==(const InputScalarType& other) const
{
  return MaskType(*static_cast<ConstBaseClass*>(this) ==
      ConstBaseClass(other));
}
```

References

[1] R.D. Sandberg, V. Michelassi, Fluid dynamics of axial turbomachinery: Blade- and stage-level simulations and models. Annual Review of Fluid Mechanics **54**(1), 255–285 (2022). doi:https://doi.org/10.1146/annurev-fluid-031221-105530

[2] K.C. Hall, J.P. Thomas, W.S. Clark, Computation of unsteady nonlinear flows in cascades using a harmonic balance technique. AIAA Journal **40**(5), 879–886 (2002). doi:https://doi.org/10.2514/2.1754

[3] C. Frey, G. Ashcroft, H.P. Kersken, C. Voigt, (2014). doi:https://doi.org/10.1115/GT2014-25230

[4] O. Krzikalla, A. Rempke, A. Bleh, M. Wagner, T. Gerhold, in New Results in Numerical and Experimental Fluid Mechanics XIII, ed. by A. Dillmann, G. Heller, E. Krämer, C. Wagner (Springer International Publishing, Cham, 2021), pp. 635–645

[5] M. Kretz, Extending c++ for explicit data-parallel programming via simd vector types. Ph.D. thesis (2015)

[6] M.S. McMullen, The application of non-linear frequency domain methods to the Euler and Navier-Stokes equations. Phd thesis, Stanford University (2003)

[7] D.A. Di Pietro, A. Ern, Mathematical aspects of discontinuous Galerkin methods, Mathématiques et Applications, vol. 69 (Springer, Heidelberg Dordrecht London New York, 2011)

[8] B. Rivière, Discontinuous Galerkin methods for solving elliptic and parabolic equations: theory and implementation. FRONTIERS IN APPLIED MATHEMATICS (Society for Industrial and Applied Mathematics, Philadelphia, 2008)

[9] G. Hager, G. Wellein, Introduction to High Performance Computing for Scientists and Engineers (CRC Press, 2010). doi:https://doi.org/10.1201/ebk1439811924

[10] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM **52**(4), 65–76 (2009). doi:https://doi.org/10.1145/1498765.1498785

[11] J. Treibig, G. Hager, G. Wellein, in 2010 39th International Conference on Parallel Processing Workshops (IEEE, 2010), pp. 207–216. doi:https://doi.org/10.1109/icppw.2010.38

[12] N. Kroll, M. Abu-Zurayk, D. Dimitrov, T. Franz, T. Führer, T. Gerhold, S. Görtz, R. Heinrich, C. Ilic, J. Jepsen, J. Jägersküpper, M. Kruse, A. Krumbein, S. Langer, D. Liu, R. Liepelt, L. Reimer, M. Ritter, A. Schwöppe, J. Scherer, F. Spiering, R. Thormann, V. Togiti, D. Vollmer, J.H. Wendisch, DLR project Digital-X: towards virtual aircraft design and flight testing based on high-fidelity methods. CEAS Aeronautical Journal **7**(1), 3–27 (2016). doi:https://doi.org/10.1007/s13272-015-0179-7. URL https://doi.org/10.1007/s13272-015-0179-7

[13] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 4.0 (2021). URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[14] T. Alrutz, J. Backhaus, T. Brandes, V. End, T. Gerhold, A. Geiger, D. Grünewald, V. Heuveline, J. Jägersküpper, A. Knüpfer, O. Krzikalla, E. Kuegeler, C. Lojewski, G. Lonsdale, R. Müller-Pfefferkorn, W. Nagel, L. Oden, F.J. Pfreundt, M. Rahn, J.P. Weiss, GASPI - A Partitioned Global Address Space Programming Interface (2013), pp. 135–136. doi:https://doi.org/10.1007/978-3-642-35893-7_18

[15] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, M. Bussmann, in High Performance Computing, ed. by J.M. Kunkel, R. Yokota, M. Taufer,

J. Shalf (Springer International Publishing, Cham, 2017), pp. 496–514

[16] H. Stengel, J. Treibig, G. Hager, G. Wellein, in Proceedings of the 29th ACM on International Conference on Supercomputing - ICS '15 (ACM Press, 2015). doi:https://doi.org/10.1145/2751205.2751240