Three-precision algebraic multigrid on GPUs

Yu-Hsiang Mike Tsai [a,*], Natalie Beams [b], Hartwig Anzt [b,a]
[a] *Karlsruhe Institute of Technology, Kaiserstraße 12, 76131, Karlsruhe, Germany*
[b] *Innovative Computing Laboratory, University of Tennessee, 1122 Volunteer Blvd, 37996 TN, Knoxville, USA*

A B S T R A C T

Recent research has demonstrated that using low precision inside some levels of an algebraic multigrid (AMG) solver can improve performance without negatively impacting the AMG quality. In this paper, we build upon previous research and implement an AMG that can use double, single, and half precision for the distinct multigrid levels. The implementation is platform-portable across GPU architectures from AMD, Intel, and NVIDIA. In an experimental analysis, we demonstrate that the use of half precision can be a viable option in multigrid. We evaluate the performance of different AMG configurations and demonstrate that mixed precision AMG can provide runtime savings compared to a double precision AMG.

Keywords:
Algebraic multigrid
Mixed precision
Portability
GPUs

1. Introduction

Finite element simulations drive a large portion of computer-assisted research and development. The principle behind finite element simulations is the discretization of the weak form of the governing partial differential equation(s) on a mesh and the computation of solution approximations through local basis functions on each mesh element. Computing these approximations requires the solution of a linear system that reflects the contributions of the local basis functions to the global solution. Given the properties of the finite element discretization, the local solutions are only directly coupled to their neighboring mesh elements, resulting in linear systems that are typically large and sparse. For one-dimensional finite element simulations, direct sparse linear solvers based on Gaussian elimination are generally a good choice, but for two- and three-dimensional simulations, the large fill-in occurring during the factorization makes direct solvers unattractive. Instead, iterative solvers generating a sequence of successively-better solution approximations are often preferred.

A successful iterative solver must find a balance between how quickly each iteration can be performed and how many iterations are required to reach the desired accuracy tolerance. A simple iterative process may be very cheap to apply on a per-iteration basis but do a poor job in terms of reducing some (or all) frequencies of the error with each iteration. This motivates the concept of multigrid methods [1–4], which use a hierarchy of successively smaller problems (*coarse grids*), related through operators that *restrict* from fine to coarse and *prolongate* back. These different levels of refinement allow the same iterative solvers to target different frequencies of the total error at the same time, based on how the error is represented on each grid. Traditionally, one distinguishes two classes of multigrid methods: those that use the geometric mesh information to derive the hierarchy of grids are called “Geometric Multigrid Methods” (GMG); those that derive the hierarchy of grids exclusively from the large sparse matrix are called “Algebraic Multigrid Methods” (AMG). AMG methods are particularly effective for discretizations of elliptic PDEs on unstructured grids and problems that lack an underlying geometric mesh.

Mixed precision methods for numerical computing have a long history and are still an active area of research [5]. With the widespread use of GPU accelerators in scientific computing, mixed precision methods exploiting their lower (single, half, or other formats like bfloat16) precision capabilities while retaining suitable accuracy are of particular interest recently [6]. When using mixed precision in the context of solving linear systems, perhaps the most common choice is a form of iterative refinement (IR). Originally proposed as a way to reduce accumulated round-off errors [7,8], it is a framework in which the solution is successively refined through corrections found by solving a linear system with the residual of the previous iteration as the right-hand side. These “inner” solves – taking place inside the outer iterative process – can often be done at a lower precision than the calculation of the residual vectors and updates to the solution. This offers the opportunity to accelerate the time to solution, as has been demonstrated on both CPUs [9,10] and GPUs [11,12].

Previous mixed precision multigrid work on GPUs has generally focused on using a reduced or mixed precision MG cycle in the inner solve of a higher-precision IR process [13–15]. This strategy is given careful theoretical consideration by Tamstorf et al. in [16,17]. In most cases, the multigrid itself is not mixed

* Corresponding author.
E-mail address: yu-hsiang.tsai@kit.edu (Y.-H.M. Tsai).

precision, meaning that the entire MG solver uses the lower precision, rather than combining different precisions on the different levels. Oo and Vogel [15] consider several configurations of mixed precision GMG for the inner solve of IR, with up to three different precisions on different levels, including half precision. They try both "directions" for the lower precision use: on the coarsest grids, with higher precisions on the finer grids, and vice versa. The former is similar in spirit to our three-precision configurations. While not a GPU implementation, a notable work of mixed precision MG apart from an IR framework is Buttari et al. [18], where the authors deploy single precision and block low rank factorization to decrease the coarse grid solution time of a very large, distributed GMG solver. In our current work, we also do not use lower or mixed precision multigrid inside iterative refinement, opting instead to test its use as a "drop in" replacement for any other double precision preconditioner in a conjugate gradient (CG) solver. We also briefly consider the performance of mixed precision AMG as a stand-alone solver in Section 4.4.

Compared to existing literature, this paper presents the following novel contributions:

1. We evaluate the use half precision within AMG with respect to preconditioner accuracy and CG convergence;
2. We demonstrate that using half precision for some AMG components can be a viable option even if aiming for high-accuracy solutions;
3. We compare the performance of mixed precision AMG on AMD MI250X, Intel PVC, and NVIDIA H100 GPUs.

The rest of the paper is structured as follows: Following some brief background in Section 2, details of the design of GINKGO's AMG, showcasing its flexibility in terms of mixed precision configuration, are given in Section 3. Section 4 evaluates the effectiveness and performance of the mixed precision AMG when used as a preconditioner inside a CG iterative solver on NVIDIA, AMD, and Intel GPUs. We then briefly discuss using mixed precision AMG as a standalone solver and the potential of mixed precision in other types of multigrid cycles. The findings are summarized in Section 5.

2. Background on AMG

Algebraic multigrid (AMG) [2,4] is a popular choice for solving or preconditioning linear problems originating from finite element discretizations. Unlike geometric multigrid (GMG), which relies on using information about the underlying geometric mesh, an AMG solver is constructed directly from the sparse system matrix. Similarly to GMG, AMG builds a hierarchy of consecutively-coarser grids and computes error correction terms on the coarser grids to improve the solution on finer grids. After creating the hierarchy, we can use it for a multigrid method like the V-Cycle in Algorithm 1. It restricts the residual on a fine grid to a coarser grid, then uses the coarser grid to obtain an error correction that is prolongated back to the finer grid to update the solution approximation. These correction computations generally entail a few iterations of an iterative method, called a "smoother" because it acts to smooth the high-frequency errors on the scale of that grid, while the coarsest grid may opt for a direct solve of the restricted problem, which is much smaller than the original matrix.

Our previous work [19] developed an AMG implementation as part of the GINKGO library that allows the user to employ single precision for coarse (i.e., not the finest) multigrid levels. We now present an extension of this work, enabling half precision computations. This results in the first AMG implementation that can employ double, single, and half precision computations for the distinct multigrid levels on AMD, Intel, and NVIDIA GPUs. We demonstrated in our first paper that GINKGO AMG can be competitive with existing open source AMG implementations – namely, NVIDIA's AmgX [20] and Lawrence Livermore National Laboratory's HYPRE [21] – on an NVIDIA GPU. In this work, we focus on the evaluation of different precision configurations, including the use of half precision, within GINKGO AMG.

Algorithm 1 V-cycle multigrid method. We use blue, red, and brown colors, respectively, to indicate the precision for: matrices (A), working vectors on the fine level (r), and working vectors on the next coarsest level (g, e). The presmoothers and postsmoothers also use the working precision for computation and matrix precision for the system matrix.

```
1: procedure VCYCLE($A$, $x$, $b$)
2:     x = PreSmooth(x, b)
3:     r = b - Ax
4:     g = Restrict(r)
5:     e = zero
6:     Vcycle(Coarse, e, g)
7:     x += Prolong(e)
8:     x = PostSmooth(x, b)
9: end procedure
```

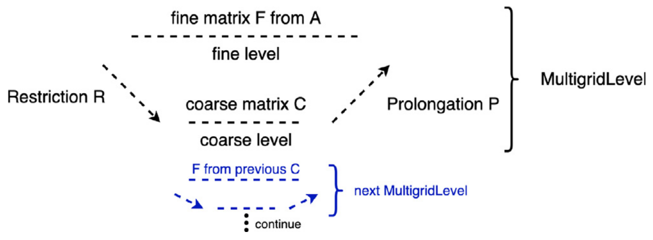


Fig. 1. The MultigridLevel class with its components.

3. Design of the flexible and platform-portable AMG

The design of the GINKGO AMG is driven by three main goals: flexibility, performance, and platform portability.

In GINKGO, we define a `MultigridLevel` class, visualized in Fig. 1, that contains a fine grid matrix from which it constructs the coarse grid matrix (C) via the coarsening algorithm, as well as the restriction (R) and prolongation (P) operators. The fine matrix (F) is the given matrix or the coarse matrix from the finer level, but with the storage precision or format potentially altered by the `MultigridLevel` according to the algorithm requirement or settings. The prolongation operator (P) is an $n \times m$ matrix built from F, and the restriction operator (R) is an $m \times n$ matrix defined as P^T. The coarse matrix (C) is an $m \times m$ matrix formed as $C = RFP$, i.e., standard Galerkin coarsening. The `MultigridLevel` class is visualized with its key components in Fig. 1.

GINKGO's AMG implementation allows the use of different precision formats on different levels of the multigrid hierarchy, resulting in a mixed precision AMG. The precision conversion happens on-the-fly in the restriction and prolongation operations. The gray and red portions of Listing 1 show a standard AMG V-cycle with a max level depth of 10, a smoother `sm` that is used for all smoothing operations, a `MultigridLevel mg_lvl`, and a coarse level solver, `coarsest_solver`. The number of smoothing sweeps is a parameter of the smoother object `sm`.

In the configuration shown in the gray and green parts of Listing 1, we enable mixed precision by adding two `Multigrid-Levels` and two smoothers in the configuration list, with `_f`

```
multigrid::build()
    .with_max_levels(10u) // equal to NVIDIA/AMGX
         11 max levels
    .with_min_coarse_row(64u)
DP  .with_pre_smoother(sm)
|   .with_mg_level(mg_lvl)
DP  .with_coarest_solver(coarest_solver)
MP  .with_pre_smoother(sm, sm_f)
||  .with_mg_level(mg_lvl, mg_lvl_f)
||  .with_coarest_solver(coarest_solver_f)
||  .with_level_selector(
||      [](const size_type level, const LinOp*)
||        -> size_type {
||          return level >= 1 ? 1 : 0;
MP      })
```

Listing 1: Configuration of a Ginkgo `Multigrid` object. Lines with a red background are used when configuring for double precision (DP), while the green background indicates configuration for mixed precision (MP).

indicating "float" or single precision. We also need to configure the `level_selector` to describe the desired scheme. Here, when the level is larger than or equal to 1 (that is, all levels except the finest grid), we use the second pair `(mg_lvl_f, sm_f)`. When the level is less than 1, we use the first pair `(mg_lvl, sm)`. Taken together, this configuration generates a mixed precision `Multigrid` where only the finest level is using double precision, and all other levels use single precision. This particular mixed precision `Multigrid` configuration allows for smooth integration as a preconditioner into an iterative solver using double precision, as the input and output vectors, as well as the original matrix, remain in double precision.

As smoother applications in the form of vector operations are relatively cheap, the runtime of an AMG cycle is generally dominated by the residual computations that involve sparse matrix–vector multiplications (*SpMV*s). We implement several common optimization techniques to reduce the overhead of these operations. Since memory allocation on the GPU is known to be expensive [22,23], we use a workspace for allocating the operator components and intermediate operations. The workspace is available over the complete lifetime of the operator.

We also reduce the overall number of residual evaluations when possible: we skip residual computation if an initial guess is zero, because we know the residual will be equal to the right-hand side vector. When using AMG as a preconditioner, we only compute the explicit residual if the user requests it, e.g. for monitoring convergence. A few optimizations regarding the handling of residual vectors are specific to the machinery of the Ginkgo library itself. For example, if we need the residual in the AMG solver, and an internal component already computed it, this residual is ensured to be accessible from outside the component. We also "split" the iteration termination check within Ginkgo, such that reaching an iteration limit terminates the algorithm before the residual is computed for convergence checking purposes.

To enable both platform and performance portability, we implement the Ginkgo AMG using a backend model as described in [24,25], where we complement an algorithm skeleton invoking a sequence of subroutines with backends containing the corresponding subroutines as heavily optimized GPU kernels in the vendor-native programming languages. Specifically, we implement CUDA kernels for NVIDIA GPUs, DPC++ kernels for Intel GPUs, and HIP kernels for AMD GPUs. Instead of having three complete stand-alone AMG implementations for the distinct GPU architectures with the corresponding kernel sets, we use C++ runtime polymorphism for automatically selecting and invoking the suitable kernels when executing the AMG algorithm. This

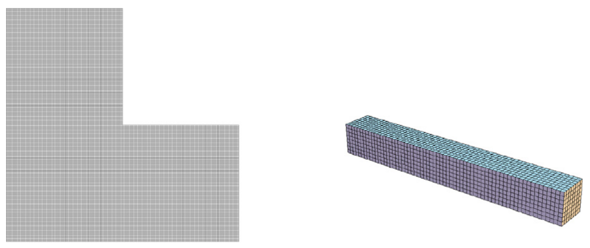


Fig. 2. Meshes used for MFEM diffusion experiments. Left: L-shape mesh with 7 levels of uniform refinement (49,152 elements); Right: Beam mesh with 3 levels of uniform refinement (4,096 elements).

allows the deployment of the AMG solver without having to maintain different variants for different hardware architectures. By doing so, we keep the cutting-edge features from vendors' official compilers without waiting for another compiler to adopt the new features.

4. Experimental evaluation

We consider a set of benchmark problems from the SuiteSparse Matrix Collection [26], as well as two diffusion problems exported from the MFEM finite element library. MFEM [27,28] is a popular open-source finite element library with support for high-order meshes and basis functions, among many other features. Our exported problems come from MFEM's "example 1", solving a standard diffusion problem $-\nabla \cdot (c\nabla u) = 1$, where c is a given coefficient. We use homogeneous Dirichlet boundary conditions. Two of MFEM's provided meshes are tested; they are shown in Fig. 2. For the "L-shape" mesh, a constant coefficient of $c = 1$ is used, while the "beam" mesh uses a piecewise constant coefficient with a jump from 1 to 0.1 at the midpoint of the length of the beam. All tests use standard third order tensor-product basis functions on the Legendre–Gauss–Lobatto nodes and MFEM's default choices for quadrature points based on the order of basis functions. A summary of all matrices is provided in Table 1, including their sizes and the range of the absolute values of all their nonzeros.

We evaluate Ginkgo's AMG implementation on GPU architectures from AMD, Intel, and NVIDIA. The GPUs and corresponding compilers are listed along with some key characteristics in Table 2. Note that the AMD MI250X has two graphics compute dies (GCDs), but the GCDs are seen individually by the system. In the following, we always consider just one GCD. Similarly, one Intel PVC GPU contains two tiles, but we run exclusively on one tile. For Intel GPUs, to our best knowledge, Ginkgo is currently the only library providing an AMG implementation with all portions of the calculation taking place on the GPU.

The experimental results are arranged as follows: first, we discuss a simple model for the potential performance gains of mixed precision AMG based on the performance of sparse matrix–vector multiplication (SpMV) operations and data storage. Next, mixed precision AMG (V-cycle) is used as a preconditioner inside a CG solver for our selected benchmark problems, and we analyze the effects on convergence rate and time per iteration. We discuss the challenges of using half precision in AMG and strategies for mitigation. Finally, we briefly consider AMG as a standalone solver and the potential for mixed precision in other multigrid cycles, like the W-cycle.

4.1. *SpMV performance as proxy for AMG*

For an idea of the potential performance benefits of mixed precision AMG on our target architectures, we analyze the performance of the sparse matrix–vector operation (SpMV), which is

Table 1
Matrix characteristics for the selected problems.

	Problem	Size	Elements	abs. value range
MFEM	beam (-o3 -l3)	120,625	14,070,001	8.9×10^{-7}–1.0
	L-shape (-o3 -l7)	443,905	11,066,881	4.0×10^{-3}–6.0
SuiteSparse	2cubes_sphere	101,492	1,647,264	6.7×10^{-15}–2.5×10^{10}
	thermal2	1,228,045	8,580,313	1.7×10^{-7}–4.9
	cage14	1,505,785	27,130,349	0.011–0.94
	cage13	445,315	7,479,343	0.012–0.93
	offshore	259,789	4,242,673	7.2×10^{-21}–7.5×10^{14}
	tmt_sym	726,713	5,080,961	8.5×10^{-14}–19

Table 2
GPU characteristics.

GPU	Peak Perf. (DP)	Peak Perf. (SP)	Mem. size	Bandwidth	Compiler
`AMD MI250X`[a] `(1 GCD)`	24 TFLOP/s	24 TFLOP/s	64 GB	1.6 TB/s	HIP 5.3
`NVIDIA H100`[b] `(PCIE)`	26 TFLOP/s	51 TFLOP/s	80 GB	2.0 TB/s	CUDA 12.0
`Intel PVC`[c] `(1 Tile)`	22.8 TFLOP/s	22.8 TFLOP/s	64 GB	1.6 TB/s	DPC++ 2023.1

Note: PVC is not released as of this writing, so the official characteristics may be changed when released.
[a]From *Frontier*, Oak Ridge National Laboratory, USA.
[b]From *BwUniCluster 2.0*, bwHPC, Germany.
[c]From *Sunspot*, Argonne National Laboratory, USA.

used in residual computations. We expect the residual computations on each level to be the major limiting factor of performance; in comparison, the solution phase of our smoothers, as well as the restriction and prolongation operations, are simpler. We extract the original matrices and the coarse matrices from levels 1–10 in an 11-level multigrid. After 2 warmup applications, we average the results from 10 SpMV evaluations. The SpMV is a memory-bound operation, with performance tied to the amount of memory traffic required rather than the number of floating point operations performed. The memory involved in a SpMV operation of an $n \times n$ CSR matrix with nnz non-zero elements can be written as:

$$(nnz + n + 1) \times I + (nnz + 2n) \times V.$$

I and V are the size, in bytes, of the IndexType and ValueType used for the CSR matrix storage. When $nnz \gg n$, the storage is approximated as $nnz \times (I + V)$. Thus, in terms of reduced memory traffic, the speedup of a lower precision ValueType V_2 over higher precision V_1 can be estimated as

$$\frac{nnz(I + V_1)}{nnz(I + V_2)} = \frac{I + V_1}{I + V_2}.$$

For example, the speedup of single precision SpMV over double precision SpMV would be 1.5x, and the speedup of half precision SpMV over double precision SpMV would be 2x.

Ginkgo has recently introduced native half precision support. The classical CSR SpMV kernel in Ginkgo is shown in Listing 2. In Fig. 4 we visualize the speedup of single and half precision SpMVs over double precision on the H100 GPU for the matrices of the distinct multigrid levels for each of the test matrices. The matrix size decreases for coarser multigrid levels, numbered with higher level numbers, as shown in Fig. 3. For the classical CSR implementation, the performance difference between the single and half precision SpMVs is negligible on H100. A possible explanation is that NVIDIA GPUs are optimized for each thread accessing at least 4 bytes of memory, which forms a 128 byte cache line. To adjust for this, we consider a "packed" version of CSR SpMV. The implementation, shown in Listing 3, processes two half precision values in one memory access. With the packed variant, the performance of the half precision SpMV generally outperforms that of single precision on H100 in Fig. 5, especially for larger cases, which is in the left side in the plots. We note that speedup values can exceed expectations due to cache effects. Figs. 6 and 7 combine the performance data and arrange the

```
template <int subwarp_size>
classical_csr_spmv(row_ptrs, col_idxs, val, b, c)
    {
    // create a subwarp with the subwarp size
    auto subwarp_tile = ...;
    // get the index of thread in a subwarp
    const auto subid = ...;
    // get the processing row index
    auto row = ...;
    const auto ind_end = row_ptrs[row + 1];
    ValueType temp_val = zero<ValueType>();
    // each thread accumulates the result; switch
         next window by subwarp_size
    for (auto ind = row_ptrs[row] + subid; ind <
        ind_end; ind += subwarp_size) {
        temp_val += val[ind] * b[col_idxs[ind]];
    }
    // use shuffle to get the summation of all
        threads in a subwarp
    auto subwarp_result = ...;
    if (subid == 0) {
        // write the result to memory
        c[row] = subwarp_result;
    }
}
```

Listing 2: Classical CSR SpMV as implemented in Ginkgo

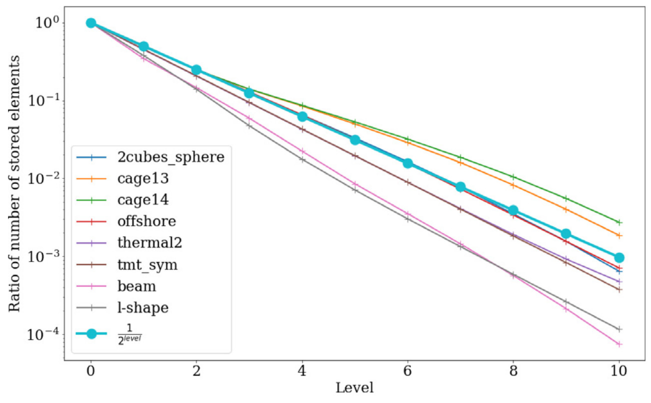


Fig. 3. The ratio of the number of stored elements in each level of a multigrid hierarchy to that of the finest level (level 0), using Parallel Graph Match [20] with size 2 for aggregation.

```
 1 template <int subwarp_size>
 2 classical_csr_spmv_pack(row_ptrs, col_idxs, val,
       b, c) {
 3     // create a subwarp with the subwarp size
 4     auto subwarp_tile = ...;
 5     // get the index of thread in a subwarp
 6     const auto subid = ...;
 7     // get the processing row index
 8     auto row = ...;
 9     const auto ind_end = row_ptrs[row + 1];
10     ValueType temp_val = zero<ValueType>();
11     // each thread accumulates the result; switch
           next window by subwarp_size*2
12     // each thread handles two elements for half
13     for (auto ind = row_ptrs[row] + subid * 2;
            ind < ind_end; ind += subwarp_size * 2)
            {
14         temp_val += val[ind] * b[col_idxs[ind]];
15         if (ind + 1 < ind_end) {
16             // Add the next element if available
17             temp_val += val[ind+1] * b[col_idxs[
                   ind+1]];
18         }
19     }
20     // use shuffle to get the summation of all
         threads in a subwarp
21     auto subwarp_result = ...;
22     if (subid == 0) {
23         // write the result to memory
24         c[row] = subwarp_result;
25     }
26 }
```

Listing 3: Classical CSR SpMV: packed variant with altered memory access

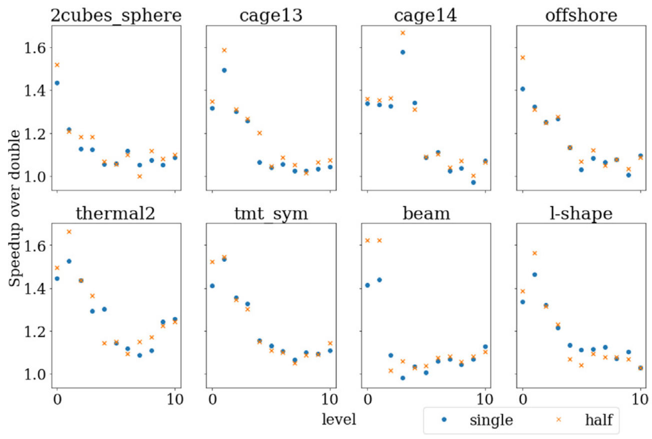


Fig. 4. H100 speedup of single and half precision SpMVs compared to double precision for the matrices produced on each level of AMG.

matrices according to increasing nonzero count. Fig. 7 shows clear improvement for the packed half variant compared to Fig. 6. For both versions, we see noticeable speedup for matrices containing more than 2e5 elements. The additional condition in line 15 of Listing 3 is only known at runtime, and is not based solely on the index of the thread, which poses a challenge for the compiler. This additional complexity compared to the classical SpMV kernel could be a factor in the cases where the packed variant performs worse than single precision, particularly since this happens for the smaller matrices. With less total memory movement involved, there is less potential speedup to cover for the effects of the extra conditional statement.

We repeat the same experiments on one GCD of an AMD MI250X GPU. Here, the original CSR implementation achieves higher performance in half precision than in single precision,

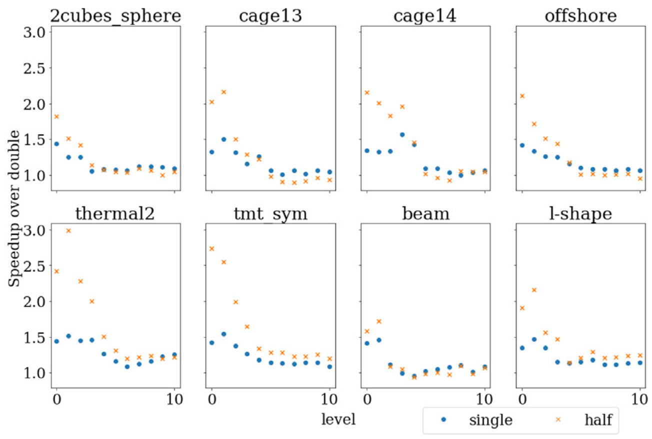


Fig. 5. H100 speedup of single and half precision SpMVs compared to double precision for the matrices produced on each level of AMG. The half precision SpMV uses the “packed half” variant.

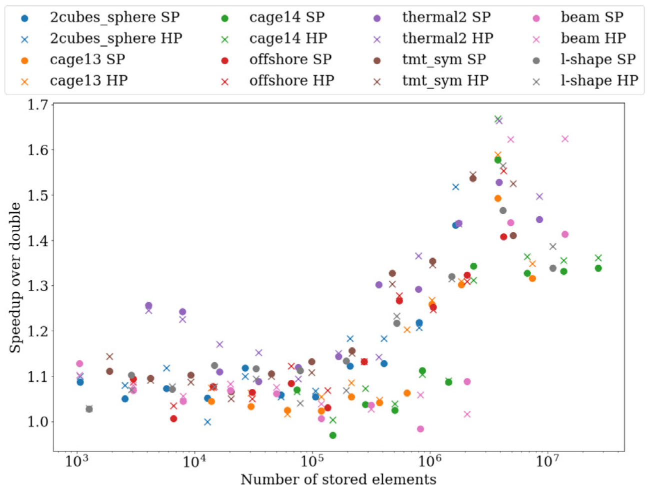


Fig. 6. H100 speedup of single (∘) and half precision (×) SpMVs for all matrices in the AMG hierarchies, arranged by total number of nonzeros.

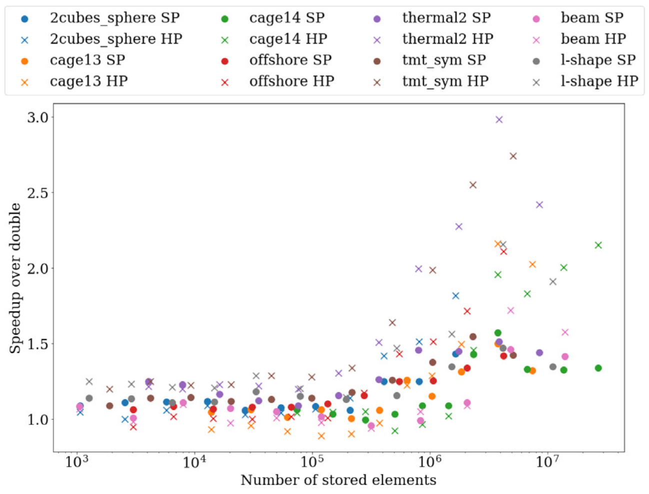


Fig. 7. H100 speedup of single (∘) and half precision (×) SpMVs for all matrices in the AMG hierarchies, arranged by total number of nonzeros. The half precision SpMV uses the “packed half” variant.

shown in Fig. 8, and the packed variant does not improve performance compared to the original version. Similar to the NVIDIA

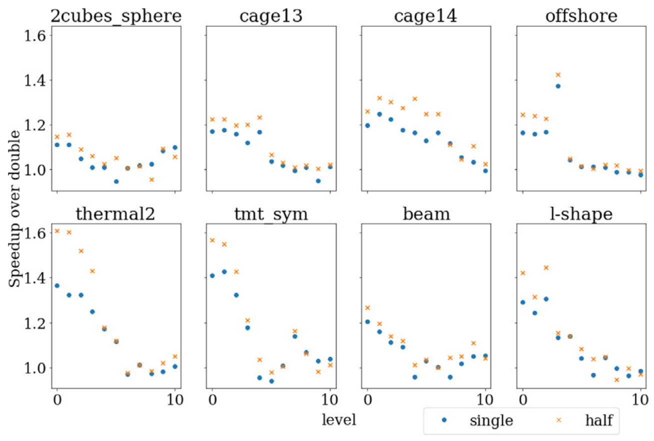


Fig. 8. MI250X speedup of single and half precision SpMVs compared to double precision for the matrices produced on each level of AMG.

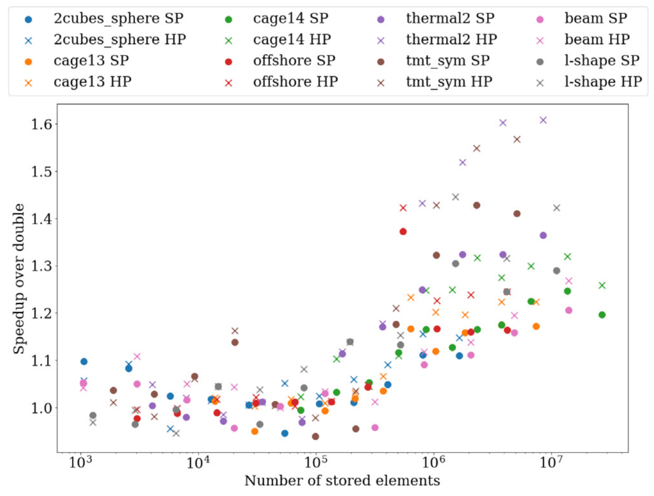


Fig. 9. MI250X speedup of single (○) and half precision (×) SpMVs for all matrices in the AMG hierarchies, arranged by total number of nonzeros.

H100 experiments, we observe significant performance benefits only for matrices containing more than 2e5 elements in Fig. 9.

Like the H100, using the packed variant for half precision gives better performance than the original version for CSR SpMV on one tile of PVC in Figs. 10 and 11. The finest matrix of cage14 can reach around 6x speedup for half precision compared to double precision. This may be related to Ginkgo's choice of `subwarp_size` for half precision compared to double precision. In Ginkgo, the subwarp size (1, 16, or 32) is chosen according to the average number of non-zeros per rows in the matrix for PVC. Intel PVC only supports 16 and 32 for subgroup (analogous to a warp in NVIDIA terminology) size, and Intel does not support the same sub-subgroup features as subwarp in CUDA currently. For the SpMV kernel on PVC, Ginkgo uses the subgroup as a subwarp safely because there is no communication out of the subgroup/subwarp. Size 1 is still available for PVC because it corresponds to one thread per row. Because threads handle two matrix elements each in half precision, the kernel can assign a smaller subwarp size in half precision when the average number of non-zeros per row is less than 32, and this smaller size may perform particularly well for some matrices. However, half precision is slower than single precision for several matrices. Moreover, the single precision speedup is quite close to 1. In this case, the kernel may use too large of a subwarp size for the short rows such that we do not fully utilize the GPU. The CUDA and

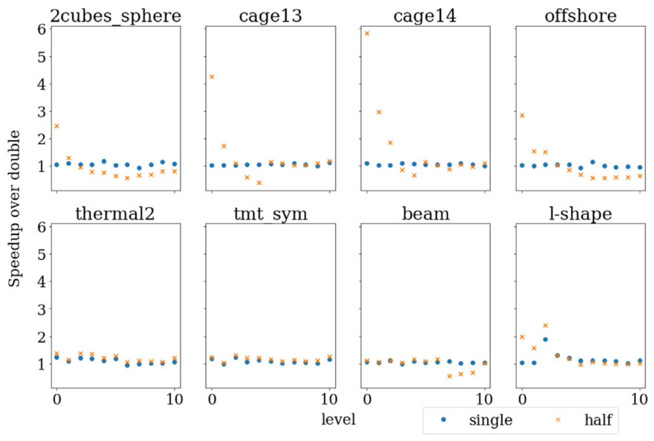


Fig. 10. PVC speedup of single and half precision SpMVs compared to double precision for the matrices produced on each level of AMG. The half precision SpMV uses the "packed half" variant.

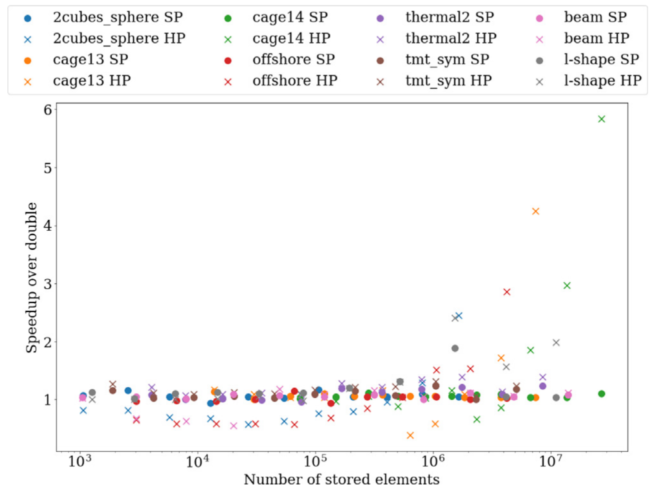


Fig. 11. PVC speedup of single (○) and half precision (×) SpMVs for all matrices in the AMG hierarchies, arranged by total number of nonzeros. The half precision SpMV uses the "packed half" variant.

HIP backends may select subwarp sizes of 2, 4, or 8, while PVC is limited to choosing between 1 and 16 for short rows, due to the limitations mentioned above.

4.1.1. Extension to mixed precision AMG

We can compute storage approximations for an entire AMG cycle based on the number of SpMVs on each level. In our implementation, coarsening usually aggregates two nodes together, while the exact compression ratio depends on the sparsity pattern. Fig. 3 reports the relative nonzero count of the matrices in the distinct AMG levels in comparison to a compression ratio of 2. Accumulating over an AMG hierarchy of $N+1$ levels, we have

$$\sum_{i=0}^{N} C_i \frac{1}{2^i} \{(nnz + n + 1) \times I + (nnz + 2n) \times V_i\}$$

$$\approx \sum_{i=0}^{N} C_i \frac{nnz \times (I + V_i)}{2^i},$$

where V_i is the size of the precision on level i and C_i is a constant determined by the total number of SpMVs performed on that level. For example, in the Jacobi smoother V-cycle configuration below, we have $C_i = 2$ for all but the coarsest level, coming from one pre- and one post-smoother application on those levels. From here, we can compute an estimate for the potential speedup of the mixed precision configuration as

$$\frac{\sum_{i=0}^{N} C_i \frac{(I+V)}{2^i}}{\sum_{i=0}^{N} C_i \frac{(I+V_i)}{2^i}}.$$

Extending the work of [19] to also include the use of half precision, we begin with four mixed precision settings:

1. (DP): all levels use double precision.
2. (DP-SP): The first level uses double precision, and all coarser levels use single precision.
3. (DP-SP-HP): The first level uses double precision, the second level uses single precision, and all coarser levels use half precision.
4. (DP-HP): The first level uses double precision, and all coarser levels use half precision.

We call these settings "uniform level", meaning that the matrix and vector working precision (marked in blue and red in Algorithm 1) are always the same on a particular level. For the (DP-SP) and (DP-HP) configurations described here, the SpMV-based speedup estimate would predict a speedup of 1.2x and 1.3x over double precision AMG, respectively.

4.2. Challenges of half precision in AMG

We first consider a straightforward continuation of the AMG experiments in [19]. For defining the restriction operators, the AMG implementation in Ginkgo uses parallel graph match (PGM), which was introduced by Naumov et al. [20] as a GPU-based algorithm for deriving a coarse approximation through exploration of the graph representation of a matrix. It is a type of aggregation method, in which nodes in the fine grid are combined to form a single coarse grid node, and can efficiently operate on sparse matrices stored in the CSR format. In Ginkgo, we deploy PGM with deterministic aggregation of size 2, meaning PGM will always try to aggregate two adjacent nodes.

The maximum number of multigrid levels is 11, with a minimum of 64 rows in the coarsest matrix. We set the stopping criterion as implicit relative residual norm reduction of 10^{-12} or maximum of 700 iterations. The pre-/post-smoothing is weighted scalar Jacobi with a weight of 0.9, i.e., $x_{i+1} = x_i + 0.9D^{-1}(b - Ax_i)$ where x_i is the solution at iteration i, D is the diagonal matrix of A, and b is the right-hand side of the linear system being solved. The same relaxation is used on the coarse grid problem, but with four relaxation sweeps instead of one.

We refer to Algorithm 1 to describe the different precision usage in the implementation. The right-hand side b and solution x are from other solvers or user inputs, so their precision is determined outside of the multigrid cycle in Algorithm 1. Residual r (red) is based on the current level's working precision. After restricting r, we store the result g and zero initial guess e (brown) in the working precision of the next coarsest level. Precision conversion is part of the restriction and prolongation operations. As mentioned previously, the matrix precision (blue) uses the same as the current level's working precision in "uniform levels".

We collect all data by performing 2 warmup applications, followed by 5 evaluation applications, of the AMG-preconditioned CG solver; the times of the 5 evaluation runs are averaged. The results are summarized in Table 3. In (DP-SP), we see similar trends as reported in [19], where the solver retains the same convergence behavior but has some improvement in performance. In (DP-SP-HP) and (DP-HP), some experiments fail to converge and some matrices see delayed convergence. (DP-SP-HP) for cage13

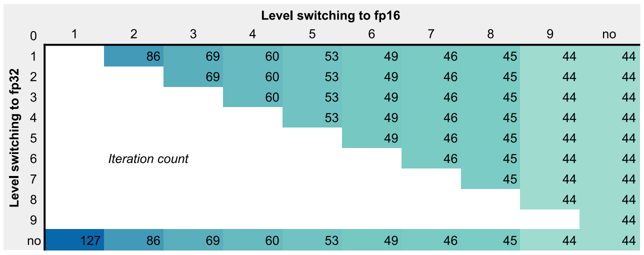


Fig. 12. H100 AMG-preconditioned CG with Jacobi smoothers: beam problem iteration counts as a function of the first level to switch to single or half precision.

and cage14 have the same number of iterations as (DP) and (DP-SP), so we can expect some performance benefit. In the beam problem, the solver using half precision exhibits a convergence delay that cannot be compensated by faster computations. Table 1 shows that the cage13 and cage14 matrices are the only ones whose non-zero values fall entirely in the range that can be represented by half precision (which is approximately 6×10^{-5}–6.55×10^{4}).

4.2.1. Effects of delayed precision changes in the AMG hierarchy

From Table 3, the effect that the use of half precision can have on convergence is clear. Using the beam problem as an example, we plot heatmaps for the number of iterations and the total time with different precision settings, delaying (in terms of distance from the finest level) the switch to single precision, half precision, or both. Each table has two axes:

- x-axis (column): The level when we change to half precision, and
- y-axis (row): The level when we change to single precision.

Each entry (x, y) means the setting is the following:

1. use double precision from the finest level (0) to the (x−1) level;
2. use single precision from the x level to (y−1) level;
3. use half precision from the y level to the coarsest level.

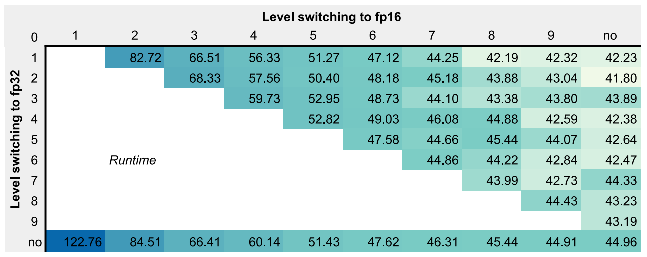
The last column ("no") only changes to single precision, and the last row ("no") only changes to half precision (skipping single precision). Also, the interaction of these two (no, no) is the purely double precision setting (DP). The coarsest solver always uses the same precision as the last level. In terms of the previously-used configurations, (no, no) = (DP), (1, no) = (DP-SP), (no, 1) = (DP-HP), and (1, 2) = (DP-SP-HP). We currently do not consider configurations where we switch to higher precision for coarser levels, though Ginkgo's configuration flexibility, demonstrated by these tests, would make this simple to consider for future numerical behavior analysis. As the iteration counts in Fig. 12 match for all configurations in the two right-most columns, the fastest configurations are those in the upper right corner, which switch to single precision sooner than those on the bottom right; see Fig. 13. Notably, these are the configurations that avoid or barely use half precision: any speedup from using half precision earlier in the multigrid cycle is not enough to make up for the extra iterations required for convergence.

4.2.2. Obstacles to convergence

Based on these initial experiments, we identified three issues affecting the AMG-preconditioned CG convergence for the matrices in Table 1 when using half precision in a "uniform precision levels" configuration:

Table 3
Performance of CG preconditioned with an AMG V-cycle, scalar Jacobi with uniform level precision configurations, on H100. The "packed half" SpMV implementation was used.

Problem	Ginkgo's AMG (DP)			Ginkgo's AMG (DP SP)			Ginkgo's AMG (DP SP HP)			Ginkgo's AMG (DP HP)		
	res. norm	#iter	Time [ms]	res. norm	#iter	Time [ms]	res. norm	#iter	Time [ms]	res. norm	#iter	Time [ms]
2cubes_sphere	6.56151e−09	20	14.0444	6.56151e−09	20	13.676	NaN	700	430.653	NaN	700	443.579
cage13	3.68272e−10	11	15.206	3.68271e−10	11	13.750	4.23527e−10	11	13.892	6.146e−10	15	17.779
cage14	3.41273e−10	10	33.494	3.41273e−10	10	30.196	3.72638e−10	10	29.433	7.02332e−10	13	35.629
offshore	1594.67	700	679.344	1325.81	700	608.188	NaN	700	629.504	NaN	700	586.523
thermal2	2.18369e−06	349	477.996	2.37802e−06	425	536.342	NaN	700	853.791	NaN	700	803.019
tmt_sym	6.94616e−05	359	360.928	7.41395e−05	401	372.497	NaN	700	645.689	NaN	700	599.100
beam-pw-sv0.1-o-3-l-3	3.05273e−15	44	45.011	3.05462e−15	44	42.417	5.52572e−15	86	79.725	7.26702e−15	127	119.101
l-shape-const-o-3-l-7	4.63552e−14	160	199.818	4.78055e−14	171	191.208	5.39855e−10	700	783.019	3.59387e−07	700	758.162



0	1	2	3	4	5	6	7	8	9	no
1		82.72	66.51	56.33	51.27	47.12	44.25	42.19	42.32	42.23
2			68.33	57.56	50.40	48.18	45.18	43.88	43.04	41.80
3				59.73	52.95	48.73	44.10	43.38	43.80	43.89
4					52.82	49.03	46.08	44.88	42.59	42.38
5						47.58	44.66	45.44	44.07	42.64
6							44.86	44.22	42.84	42.47
7								43.99	42.73	44.33
8									44.43	43.23
9										43.19
no	122.76	84.51	66.41	60.14	51.43	47.62	46.31	45.44	44.91	44.96

Fig. 13. H100 AMG-preconditioned CG with Jacobi smoothers: beam problem total solve time as a function of the first level to switch to single or half precision.

Algorithm 2 Symmetry-preserving row and column equilibration (one iteration of Algorithm 2.5 in [29]).

```
1: procedure SCALE($A$)
2:     for all $i$ in $0 \ldots n-1$ do
3:         Let $r_i = max(abs(A(i,:)))^{1/2}$
4:         Let $c_i = max(abs(A(:,i)))^{1/2}$
5:     end for
6:     for all $row, col$ in $A$ do
7:         $A(row, col) = \frac{A(row,col)}{r_{row} c_{col}}$
8:     end for
9: end procedure
```

- The matrix values are out of the range of half precision: 2cubes_sphere and offshore.
  When applying the matrix with a zero vector, $Inf * 0 = NaN$.
- The application of the Jacobi smoother is out of the range of half-precision: thermal2 and tmt_sym.
  For some diagonal values that are within the range of half precision, the inverted values are not.
- The residual is too small in coarse levels: L-shape.
  This occurs for each level of multigrid where the residual passed to the coarser level after the restriction is zero in half precision due to underflow. This means the right-hand side of the coarser level becomes zero, giving the trivial solution on that level, and thus no contribution to the overall correction. L-shape does not encounter the NaN issue, but the convergence is slower than the other settings. This issue may materialize after a few iterations, rather than from the beginning.

We utilize two mitigation strategies. First, row/column scaling as given in Algorithm 2.5 of [29], detailed in Algorithm 2, is applied to the 2cubes_sphere and offshore matrices to ensure all values are in the half precision range. To directly compare the convergence of higher precisions and mixed precision, we perform the scaling on these matrices for all precision configurations, not just those involving half. We also add a scaling of 1/2 to the restriction matrix in the aggregation method to protect against values exceeding the allowable range after the merging step. With scaling in Algorithm 2, the (DP-SP-HP) configuration on 2cubes_sphere can solve the problem without incurring NaNs.

Second, we employ the common mixed precision technique of decoupling the working vector precision from the matrix storage precision, analogous to computing IR residuals in a higher precision to combat roundoff errors. Using a higher precision for the vectors helps avoid zero residuals and Jacobi smoother application overflow, but we retain most of the benefit from half precision in SpMV as the matrix accounts for the bulk of the memory movement. This also allows us to use a lower precision to store the finest level matrix. This will incur some extra setup cost and memory usage, as it requires copying and converting the original double precision matrix to store in half precision, but increases the potential speedup for each application of the AMG cycle.

Returning to Algorithm 1, we now also have "non-uniform levels", where the matrix precision can be chosen separately from the vector/working precision. The arithmetic operations in the residual computation will always use the highest precision format among input, output, and matrix precisions and store the result in the output precision. In our configurations, the working vector precision always uses more bits than the matrix precision when they differ, so the precision used in the arithmetic operations is always the working vector precision.

We consider the following non-uniform configurations:

- The **vector precision** (also the **arithmetic precision** in our cases):
  1. (DP-SP): the first level's vector uses double precision, but the other levels' vectors use single precision.
  2. (DP): all vectors use double precision.
- The **matrix precision**:
  1. (SP): all matrices in multigrid use single precision.
  2. (HP): all matrices in multigrid use half precision.
  3. (DP-SP-HP): the first level's matrix uses double precision, the second level's matrix uses single precision, and the other levels' matrices use half precision.

Thus, we have 6 possible combinations from these options. In the following discussion and figures, we use the notation *(Working Precision, Matrix Precision)* to represent a specific precision configuration. The uniform levels from previous experiments will now be represented with the same precision shorthand repeated, e.g. (DP-HP, DP-HP).

4.2.3. Additional configuration options to improve convergence

Even in double precision, some of our test problems are difficult for the simple AMG configuration used in the previous experiments; the offshore problem did not converge within the iteration limit. To address this issue, as well as to compare mixed precision performance in AMG cycles using more complicated smoothers, we add two additional smoother configuration options. The first is block Jacobi, with all non-block-size settings identical to the scalar Jacobi smoother. We use 32 as the maximum block size for block Jacobi smoothers (see [30] for details of GINKGO's block Jacobi implementation). Block Jacobi also did not succeed in achieving convergence for the offshore problem, so we will omit the configurations based on scalar and block Jacobi for this problem. We also test a configuration using an ℓ_1-Jacobi Chebyshev smoother, combining ℓ_1-Jacobi from [31] with the Chebyshev iteration, as in [32]. For this smoother, we set 2 iterations each for pre-/post-smoothing, as well as 2 iterations for the coarse solver. As described in [32], when using ℓ_1-Jacobi as the inner solver of Chebyshev on SPD (symmetric positive definite) matrices, all eigenvalues are in $(0, 1]$. For the two non-SPD matrices – cage13 and cage14 – we omit this smoother configuration in the following tests. This smoother is able to improve the (DP) convergence of the offshore problem such that it converges before reaching the iteration limit of 700 iterations.

4.3. Comparison of uniform and non-uniform precision configurations

On the H100, the convergence benefit of the non-uniform configuration is clear. (DP-SP, HP) usually achieves the best speedup in total solve time in Figs. 14 to 16. For thermal2 and tmt_sym, we need more iterations with single precision working vectors (Figs. 17 to 19), which affects performance. With double precision working vectors on every level, we can keep the same iteration as the all-double multigrid setup, so (DP, HP) is the best for the thermal2 and tmt_sym cases. (DP-SP, HP) in tmt_sym still shows some speedup compared to the full double settings because the speedup per iteration can compensate for the increase in iterations. In Fig. 19, (DP-SP-HP, DP-SP-HP) and (DP-HP, DP-HP) configurations do not converge because ℓ_1-Jacobi's addition of the absolute values of all off-diagonal entries to the diagonal value causes overflow in half precision. We can get up to 1.35x speedup in the preconditioned CG with Jacobi smoothers, up to 1.27x speedup with block Jacobi smoothers, and up to 1.24x speedup with ℓ_1-Jacobi Chebyshev smoothers. Note that if no speedup was achieved by a particular configuration, it does not have a bar for that problem. We embed additional information into speedup figures: "NaN ($\times$)" indicates the result residual norm is NaN, "More iter ($\circ$)" indicates the configuration requires more iterations than the full double precision configuration, "No speedup per iter ($\triangle$)" indicates the time per iteration is not faster than the full double precision configuration, and "Not converged ($\triangledown$)" indicates the configuration does not converge.

The performance trends are very similar when executing on the MI250X in Figs. 20 to 22. (DP-SP, HP) is usually the fastest option with Jacobi and blockJacobi smoothers. For cage14, (DP-SP, HP) performs the best, and can get up to 1.13x speedup with Jacobi smoothers and up to 1.16x speedup with blockJacobi smoothers. In Fig. 22, we get 1.25x speedup from (DP-SP, SP) for the beam problem, but around 1.05x–1.10x speedup from (DP-SP, HP) or (DP, HP) in other cases. We expect less speedup from mixed precision than for H100 because of the smaller performance differences in the SpMV analysis from Figs. 7 and 9.

In Figs. 23 to 25, we present the results obtained from running on Intel's PVC GPU (one tile). The performance trends are different from those we observed for the MI250X and the H100. We get around 1.15x speedup for the tmt_sym problem with (DP, DP-SP-HP) and for cage13 with (DP-SP, HP), while seeing almost 1.25x speedup for the cage14 (DP-SP, HP) case with Jacobi and Block-Jacobi smoothers. In ℓ_1-Jacobi-Chebyshev, only (DP-SP, DP-SP) in

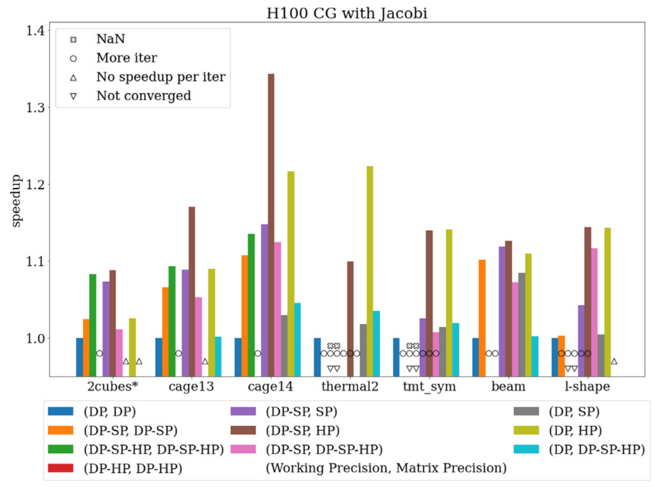


Fig. 14. Speedup in total solve time for CG with AMG V-cycle preconditioning, scalar Jacobi configuration. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

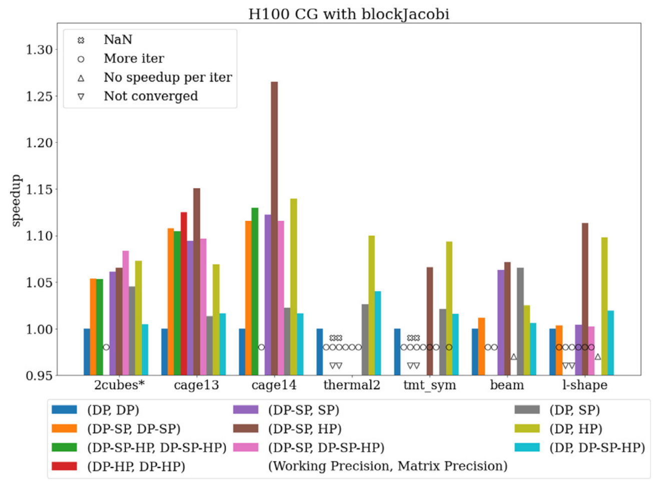


Fig. 15. Speedup in total solve time for CG with AMG V-cycle preconditioning, block Jacobi configuration. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

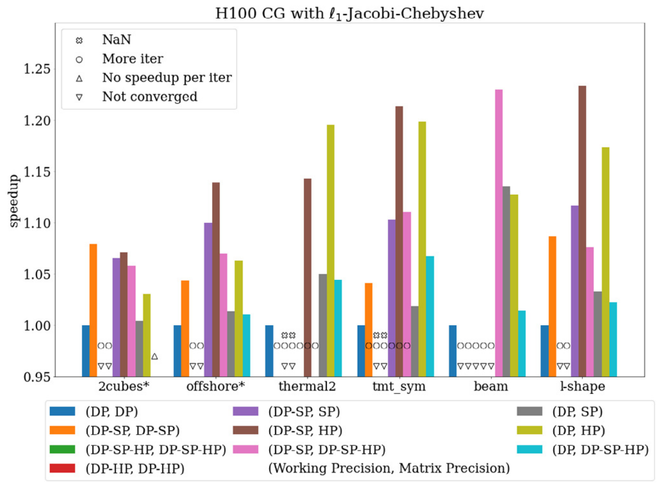


Fig. 16. Speedup in total solve time for CG with AMG V-cycle preconditioning, ℓ_1-Jacobi-Chebyshev configuration. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

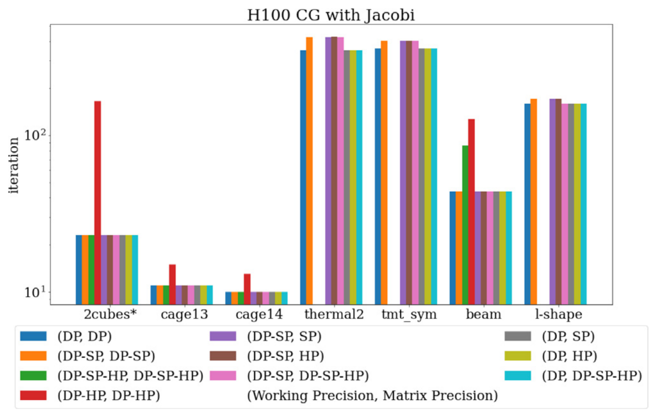


Fig. 17. Total iterations for CG with AMG V-cycle preconditioning, scalar Jacobi configuration. Results are shown for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

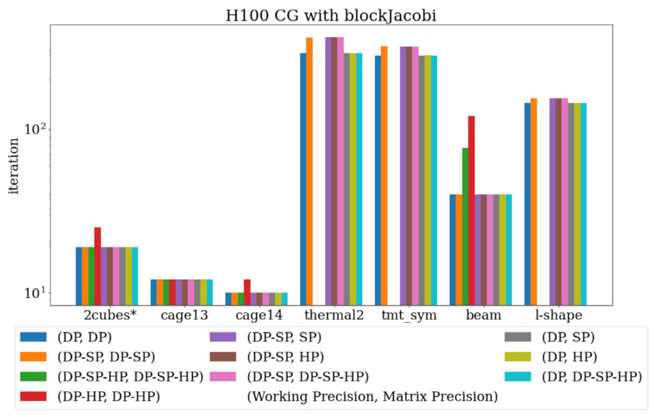


Fig. 18. Total iterations for CG with AMG V-cycle preconditioning, block Jacobi configuration. Results are shown for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

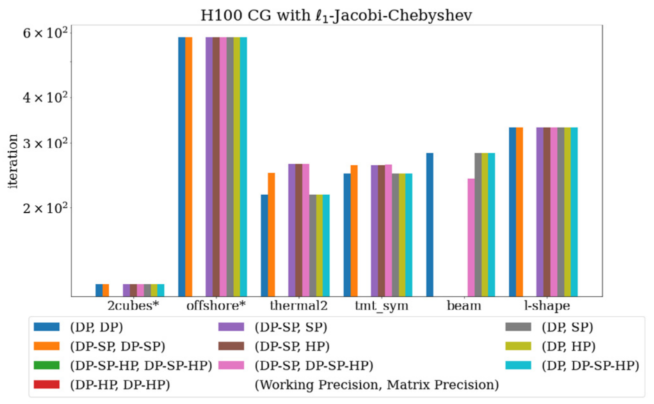


Fig. 19. Total iterations for CG with AMG V-cycle preconditioning, ℓ_1-Jacobi-Chebyshev configuration. Results are shown for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

beam and (DP-SP, HP) for L-shape get close to 1.2x speedup. The other configurations does not show much speedup with mixed precision on PVC, which is expected from the SpMV performance in Figs. 10 and 11.

Finally, in Table 4, we collect the fastest (in total solve time) configuration for each matrix. The benefit of the non-uniform

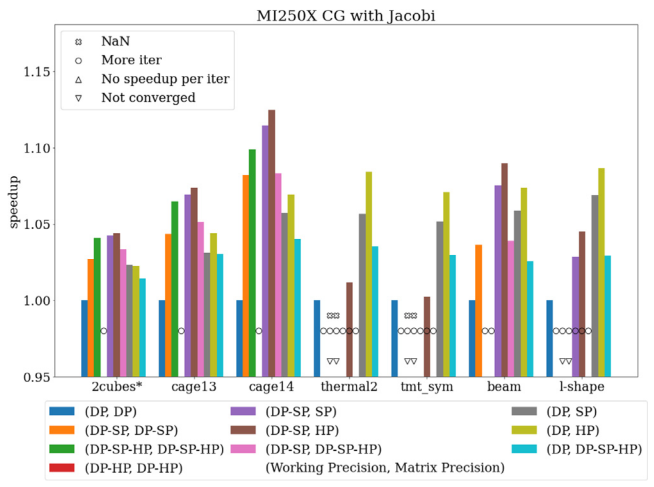


Fig. 20. Speedup in total solve time for CG with AMG V-cycle preconditioning, scalar Jacobi configuration. Results are for one GCD of MI250X. * denotes the matrix was scaled prior to solving.

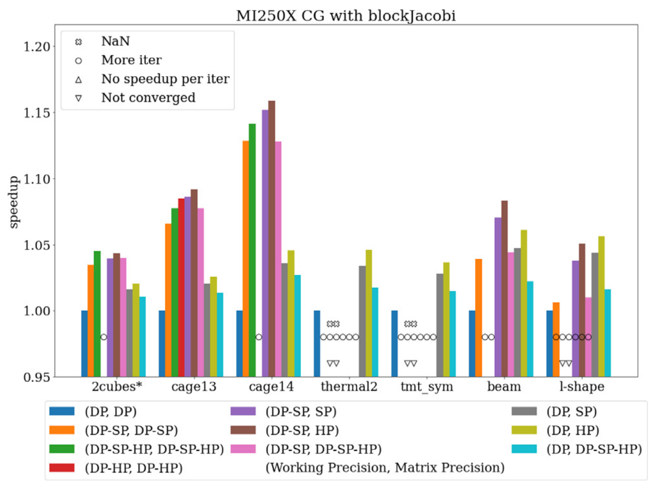


Fig. 21. Speedup in total solve time for CG with AMG V-cycle preconditioning, block Jacobi configuration. Results are for one GCD of MI250X. * denotes the matrix was scaled prior to solving.

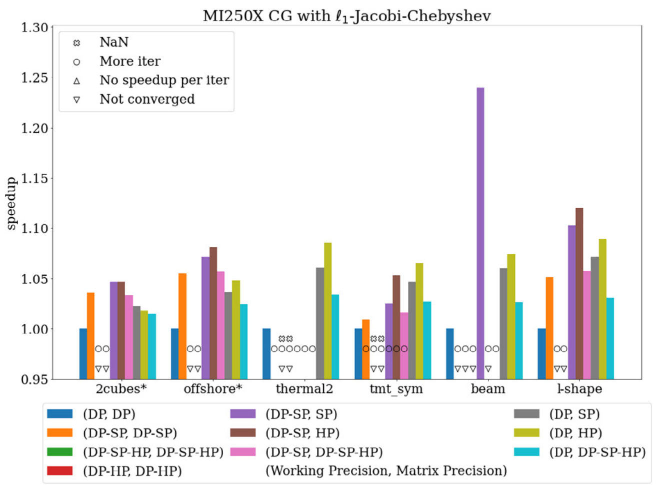


Fig. 22. Speedup in total solve time for CG with AMG V-cycle preconditioning, ℓ_1-Jacobi-Chebyshev configuration. Results are for one GCD of MI250X. * denotes the matrix was scaled prior to solving.

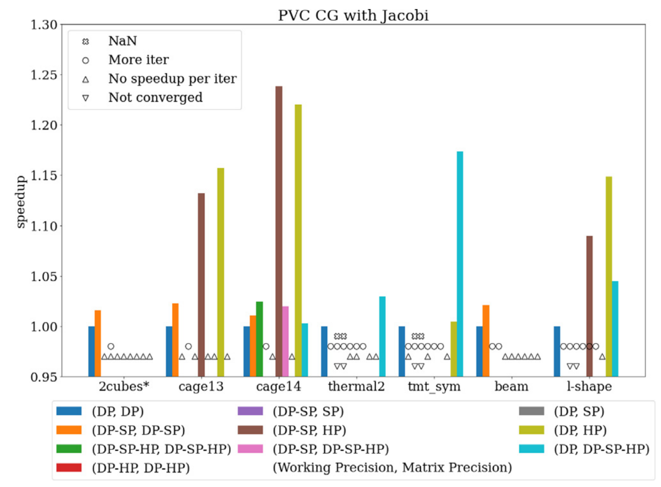


Fig. 23. Speedup in total solve time for CG with AMG V-cycle preconditioning, scalar Jacobi configuration. Results are for one tile of PVC with the packed half variant SpMV. * denotes the matrix was scaled prior to solving.

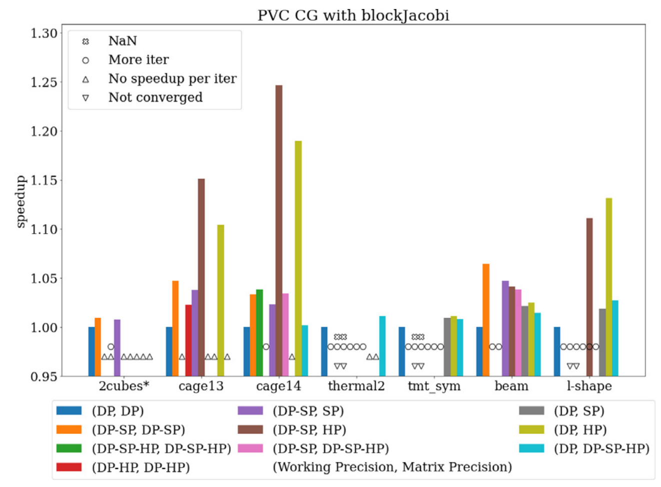


Fig. 24. Speedup in total solve time for CG with AMG V-cycle preconditioning, block Jacobi configuration. Results are for one tile of PVC with the packed half variant SpMV. * denotes the matrix was scaled prior to solving.

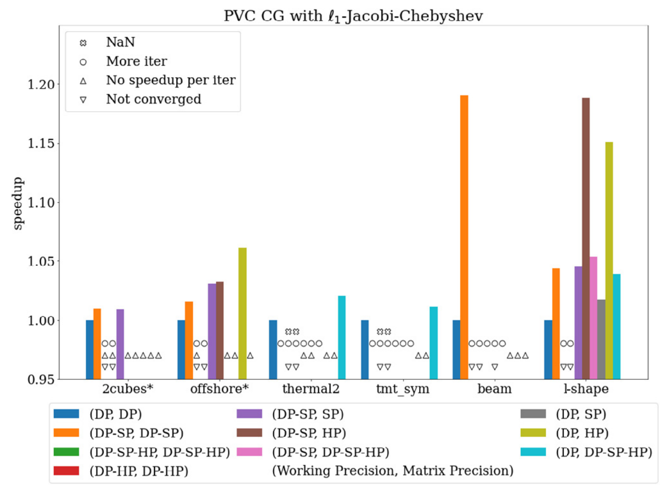


Fig. 25. Speedup in total solve time for CG with AMG V-cycle preconditioning, ℓ_1-Jacobi-Chebyshev configuration. Results are for one tile of PVC with the packed half variant SpMV. * denotes the matrix was scaled prior to solving.

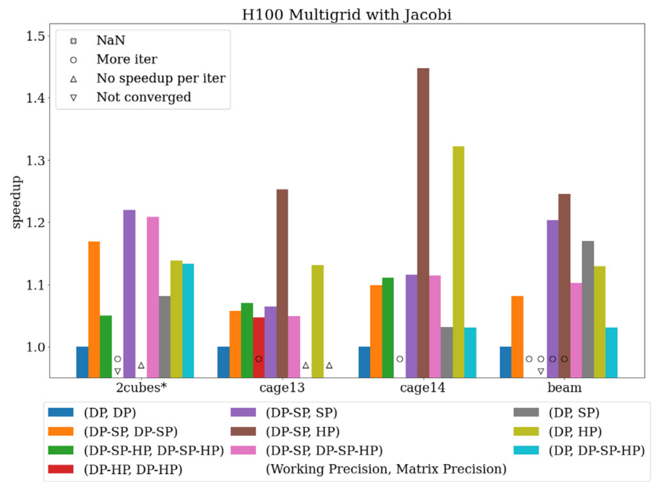


Fig. 26. Speedup in total solve time for a standalone V-cycle AMG solver (scalar Jacobi configuration) on H100, using packed half SpMV. * denotes the matrix was scaled prior to solving.

mixed precision configurations is clear across all three architectures. Notably, for each matrix, the same smoother was the best choice for all three architectures, and it was the scalar Jacobi smoother for all but the offshore matrix. In terms of specific precision configurations, MI250X and H100 always align, yet PVC is only the same for the cage14 and L-shape problems. The "winning" configuration for every problem uses half precision matrices on at least some levels, demonstrating that while the use of half precision presents more challenges than single precision, both in terms of convergence and efficient implementation, solutions like matrix scaling, the use of higher working precision, and the packed SpMV variant can help address the issues that may arise.

4.4. *Standalone multigrid solver*

We also demonstrate GINKGO's AMG as a standalone solver on H100 in Figs. 26 and 27. From the success of the non-uniform settings with half precision in the preconditioner experiments, we use the Jacobi smoother with the same parameters as before, except with the absolute residual norm stopping criterion set to 1e−9. There are four cases that converge in double precision settings (DP, DP): 2cubes_sphere, cage13, cage14, and beam. The (DP-HP, DP-HP) configuration is generally poor for convergence. This is expected from the previously-described challenges of using half precision: the 2cubes_sphere and beam problems do not converge, and cage13 and cage14 require more iterations than the other configurations. However, the non-uniform settings with higher working precision preserve convergence. A non-uniform configuration with half precision matrices performs the best for all but the 2cubes_sphere problem. Overall, the standalone multigrid shows higher speedup than the preconditioned CG because there is no additional double precision work outside of the multigrid. The cage14 problem attains up to 1.45x speedup with (DP-SP, HP) configuration. For these parameters, using the smoother in double precision only at the finest level is sufficient to preserve convergence and final accuracy without an outer CG solver or iterative refinement.

4.5. *Other multigrid cycles*

The standalone multigrid solver's greater potential for speedup with mixed precision is due to, in essence, reducing the ratio of

Table 4
The fastest configuration for AMG-preconditioned CG, in terms of total solve time, across all smoothers and all mixed precision settings.

	Smoother	H100	MI250X(1GCD)	PVC(1tile)
2cubes_sphere*	Jacobi	(DP-SP, HP)	(DP-SP, HP)	(DP-SP, DP-SP)
cage13	Jacobi	(DP-SP, HP)	(DP-SP, HP)	(DP, HP)
cage14	Jacobi	(DP-SP, HP)	(DP-SP, HP)	(DP-SP, HP)
offshore*	ℓ_1-Jacobi-Chebyshev	(DP-SP, HP)	(DP-SP, HP)	(DP, HP)
thermal2	Jacobi	(DP, HP)	(DP, HP)	(DP, DP-SP-HP)
tmt_sym	Jacobi	(DP, HP)	(DP, HP)	(DP, DP-SP-HP)
beam-pw-sv0.1-o-3-l-3	Jacobi	(DP-SP, HP)	(DP-SP, HP)	(DP-SP, DP-SP)
l-shape-const-o-3-l-7	Jacobi	(DP, HP)	(DP, HP)	(DP, HP)

The matrices marked with * were scaled to avoid exceeding the representation range of half precision.

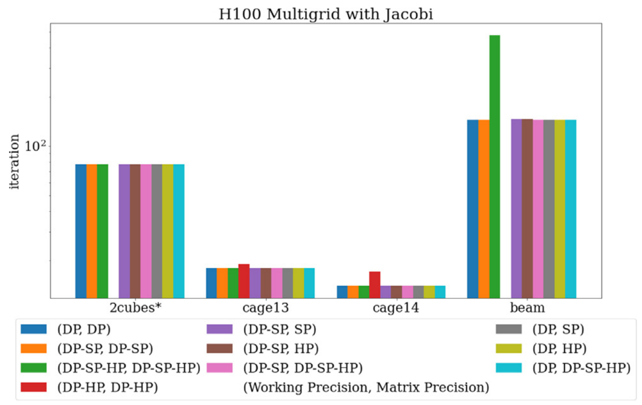


Fig. 27. Total iterations for a standalone V-cycle AMG solve (scalar Jacobi configuration) on H100, using packed half SpMV. * denotes the matrix was scaled prior to solving.

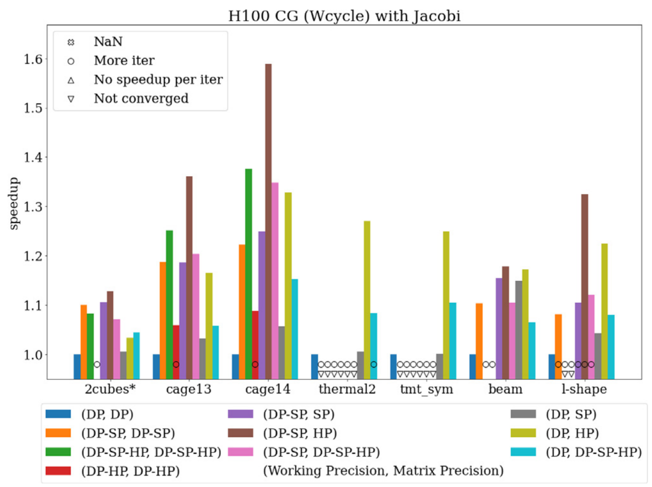


Fig. 28. Speedup in total solve time for CG with AMG W-cycle preconditioning, scalar Jacobi configuration with 4 levels. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

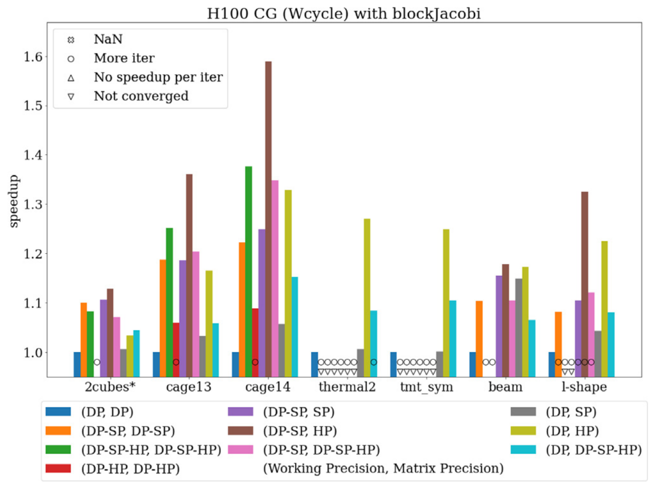


Fig. 29. Speedup in total solve time for CG with AMG W-cycle preconditioning, block Jacobi configuration with 4 levels. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

double precision use to lower precision use. To raise the "maximum speedup ceiling" while still using AMG as a preconditioner inside a higher-precision linear solver like CG, the use of cycles other than the V-cycle is an option. As we move through the grid hierarchy from finest to coarsest, each level contributes less to the total overall runtime and memory movement requirements of the AMG as the matrix sizes decrease. Thus, coarse levels which could achieve speedup "individually", i.e., relative to their corresponding levels in a fully double precision multigrid, will have a limited impact on the speedup of the *full* cycle. If we visit the coarse levels more often during one full cycle of multigrid, resulting in more operations on the coarse matrices, we decrease the ratio of double precision to lower precision for one full iteration and thereby increase the potential benefit of the mixed precision AMG. F- and W-cycles, for example, spend more time on the coarse levels during each multigrid cycle. We try the same CG experiments on a small W-cycle with 4 levels. In Figs. 28 to 30, we can get up to 1.6x speedup with Jacobi smoothers, 1.45x speedup with block Jacobi smoothers, and 1.4x speedup with ℓ_1-Jacobi Chebyshev smoothers on H100. As with the V-cycle, when separating the precision of working vectors and matrices on a level, we can use lower precision for the matrix on the finest level to enable even higher speedups.

5. Conclusion

In this paper, we implement and evaluate a mixed precision algebraic multigrid (AMG) method that allows the use of double precision, single precision, and half precision. We demonstrate that when using AMG as a preconditioner inside an iterative solver, some linear systems allow for using single precision or even half precision on the coarser multigrid levels without impacting the solution quality. We use techniques such as scaling and decoupling the precision formats for the working vectors and matrices on the same level to address convergence issues caused by the use of half precision. When using higher precision in working vectors, we found more mixed precision cases that maintain the same number of iterations as the full double precision settings

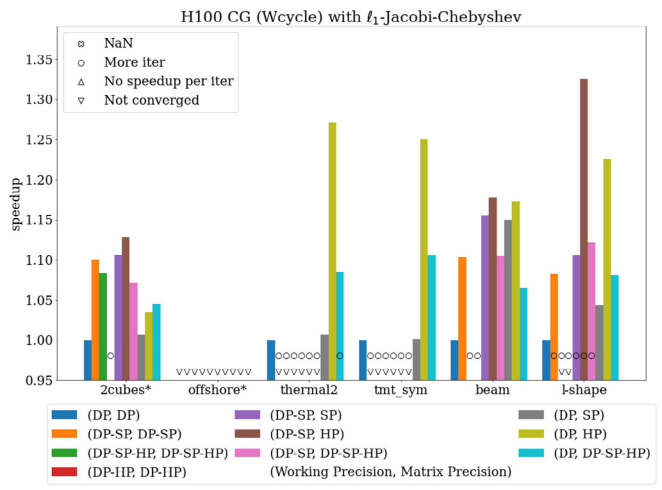


Fig. 30. Speedup in total solve time for CG with AMG W-cycle preconditioning, ℓ_1-Jacobi-Chebyshev configuration with 4 levels. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

while taking less time. Performance evaluations were completed on AMD, Intel, and NVIDIA GPUs.

CRediT authorship contribution statement

Yu-Hsiang Mike Tsai: Conceptualization, Methodology, Writing – original draft, Investigation, Formal analysis, Visualization. **Natalie Beams:** Formal analysis, Resources, Writing – review & editing. **Hartwig Anzt:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The authors acknowledge support by the state of Baden-Württemberg through bwHPC. This work was supported by the “Impuls und Vernetzungsfond” of the Helmholtz Association under grant VH-NG-1241, and the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, and resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Math. Comp. 31 (138) (1977) 333–390, http://dx.doi.org/10.1090/S0025-5718-1977-0431719-X.
[2] J.W. Ruge, K. Stüben, 4. Algebraic Multigrid, in: Multigrid Methods, in: Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, 1987, pp. 73–130, http://dx.doi.org/10.1137/1.9781611971057.ch4.
[3] P. Wesseling, C.W. Oosterlee, Geometric multigrid with applications to computational fluid dynamics, Numerical Analysis 2000. Vol. VII: Partial Differential Equations, J. Comput. Appl. Math. 128 (1) (2001) 311–334, http://dx.doi.org/10.1016/S0377-0427(00)00517-3.
[4] R. Falgout, An introduction to algebraic multigrid, Comput. Sci. Eng. 8 (6) (2006) 24–33, http://dx.doi.org/10.1109/MCSE.2006.105.
[5] N.J. Higham, T. Mary, Mixed precision algorithms in numerical linear algebra, Acta Numer. 31 (2022) 347–414, http://dx.doi.org/10.1017/S0962492922000022.
[6] A. Abdelfattah, H. Anzt, E.G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N.J. Higham, X.S. Li, et al., A survey of numerical linear algebra methods utilizing mixed-precision arithmetic, Int. J. High Perform. Comput. Appl. 35 (4) (2021) 344–369.
[7] J.H. Wilkinson, Rounding Errors in Algebraic Processes, in: Prentice-Hall series in automatic computation, Prentice-Hall, Englewood Cliffs, N.J, 1964.
[8] C.B. Moler, Iterative Refinement in Floating Point, J. ACM 14 (2) (1967) 316–321, http://dx.doi.org/10.1145/321386.321394.
[9] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Dongarra, Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems), in: SC ’06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006, p. 50, http://dx.doi.org/10.1109/SC.2006.30.
[10] E. Carson, N.J. Higham, Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions, SIAM J. Sci. Comput. 40 (2) (2018) A817–A847, http://dx.doi.org/10.1137/17M1140819.
[11] A. Haidar, S. Tomov, J. Dongarra, N.J. Higham, Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2018, pp. 603–613, http://dx.doi.org/10.1109/SC.2018.00050.
[12] J.A. Loe, C.A. Glusa, I. Yamazaki, E.G. Boman, S. Rajamanickam, Experimental Evaluation of Multiprecision Strategies for GMRES on GPUs.
[13] D. Göddeke, R. Strzodka, Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid, IEEE Trans. Parallel Distrib. Syst. 22 (1) (2010) 22–32.
[14] Y. Sumiyoshi, A. Fujii, A. Nukada, T. Tanaka, Mixed-precision AMG method for many core accelerators, in: Proceedings of the 21st European MPI Users’ Group Meeting, 2014, pp. 127–132.
[15] K.L. Oo, A. Vogel, Accelerating Geometric Multigrid Preconditioning with Half-Precision Arithmetic on GPUs, 2020, arXiv:2007.07539 [cs].
[16] S.F. McCormick, J. Benzaken, R. Tamstorf, Algebraic error analysis for mixed-precision multigrid solvers, SIAM J. Sci. Comput. 43 (5) (2021) S392–S419.
[17] R. Tamstorf, J. Benzaken, S.F. McCormick, Discretization-Error-Accurate Mixed-Precision Multigrid Solvers, SIAM J. Sci. Comput. 43 (5) (2021) S420–S447, http://dx.doi.org/10.1137/20M1349230.
[18] A. Buttari, M. Huber, P. Leleux, T. Mary, U. Rüde, B. Wohlmuth, Block low-rank single precision coarse grid solvers for extreme scale multigrid methods, Numer. Linear Algebra Appl. 29 (1) (2022) e2407, http://dx.doi.org/10.1002/nla.2407.
[19] Y.-H.M. Tsai, N. Beams, H. Anzt, Mixed Precision Algebraic Multigrid on GPUs, in: R. Wyrzykowski, J. Dongarra, E. Deelman, K. Karczewski (Eds.), Parallel Processing and Applied Mathematics, in: Lecture Notes in Computer Science, Springer International Publishing, Cham, 2023, pp. 113–125, http://dx.doi.org/10.1007/978-3-031-30442-2_9.
[20] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, et al., AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods, SIAM J. Sci. Comput. 37 (5) (2015) S602–S626.
[21] U.M. Yang, et al., BoomerAMG: a parallel algebraic multigrid solver and preconditioner, Appl. Numer. Math. 41 (1) (2002) 155–177.
[22] NVIDIA, CUDA Best Practices, 2023, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. (Accessed 14 July 2023).
[23] AMD, rocBLAS Contributor’s Guide, 2023, https://rocblas.readthedocs.io/en/rocm-5.3.0/Contributors_Guide.html, 2023-07-14.

[24] T. Cojean, Y.-H.M. Tsai, H. Anzt, Ginkgo—A math library designed for platform portability, Parallel Comput. 111 (2022) 102902, http://dx.doi.org/10.1016/j.parco.2022.102902, URL https://www.sciencedirect.com/science/article/pii/S0167819122000096.

[25] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.M. Tsai, E.S. Quintana-Ortí, Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing, ACM Trans. Math. Software 48 (1) (2022) 2:1–2:33, http://dx.doi.org/10.1145/3480935.

[26] T.A. Davis, Y. Hu, The University of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1–25.

[27] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J.C.V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, S. Zampini, MFEM: A modular finite element methods library, Comput. Math. Appl. 81 (2021) 42–74, http://dx.doi.org/10.1016/j.camwa.2020.06.009.

[28] MFEM: Modular finite element methods [Software], 2017, http://dx.doi.org/10.11578/dc.20171025.1248, mfem.org.

[29] N.J. Higham, S. Pranesh, M. Zounon, Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems, SIAM J. Sci. Comput. 41 (4) (2019) A2536–A2551, http://dx.doi.org/10.1137/18M1229511.

[30] G. Flegar, H. Anzt, T. Cojean, E.S. Quintana-Ortí, Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software, ACM Trans. Math. Softw. 47 (2) (2021) http://dx.doi.org/10.1145/3441850.

[31] A.H. Baker, R.D. Falgout, T.V. Kolev, U.M. Yang, Multigrid Smoothers for Ultraparallel Computing, SIAM J. Sci. Comput. 33 (5) (2011) 2864–2887, http://dx.doi.org/10.1137/100798806, URL https://epubs.siam.org/doi/abs/10.1137/100798806.

[32] A. El Haman Abdeselam, A. Napov, Y. Notay, Porting an aggregation-based algebraic multigrid method to GPUs, ETNA - Electron. Trans. Numer. Anal. 55 (2022) 687–705, http://dx.doi.org/10.1553/etna_vol55s687, URL https://hw.oeaw.ac.at?arp=0x003da4b8.