Scaling the memory wall using mixed-precision - HPG-MxP on an exascale machine

Aditya Kashi
National Center for Computational Sciences
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
kashia@ornl.gov

Nicholson Koukpaizan
National Center for Computational Sciences
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
koukpaizannk@ornl.gov

Hao Lu
National Center for Computational Sciences
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
luh1@ornl.gov

Michael Matheson
National Center for Computational Sciences
Oak Ridge National Laboratory (ORNL)
Oak Ridge, USA
mathesonma@ornl.gov

Sarp Oral
National Center for Computational Sciences
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
oralhs@ornl.gov

Feiyi Wang
National Center for Computational Sciences
Oak Ridge National Laboratory (ORNL)
Oak Ridge, TN, USA
fwang2@ornl.gov

Abstract

Mixed-precision algorithms have been proposed as a way for scientific computing to benefit from some of the gains seen for artificial intelligence (AI) on recent high performance computing (HPC) platforms. A few applications dominated by dense matrix operations have seen substantial speedups by utilizing low precision formats such as FP16. However, a majority of scientific simulation applications are memory bandwidth limited. Beyond preliminary studies, the practical gain from using mixed-precision algorithms on a given HPC system is largely unclear.

The High Performance GMRES Mixed Precision (HPG-MxP) benchmark has been proposed to measure the useful performance of a HPC system on sparse matrix-based mixed-precision applications. In this work, we present a highly optimized implementation of the HPG-MxP benchmark for an exascale system and describe our algorithm enhancements. We show for the first time a speedup of 1.6× using a combination of double- and single-precision on modern GPU-based supercomputers.

CCS Concepts

• **Mathematics of computing** → **Mathematical software performance**; • **Applied computing** → *Physical sciences and engineering*.

Keywords

Mixed precision, sparse linear algebra, iterative linear solver, graphics processing unit, parallel scaling, benchmark

ACM Reference Format:
Aditya Kashi, Nicholson Koukpaizan, Hao Lu, Michael Matheson, Sarp Oral, and Feiyi Wang. 2025. Scaling the memory wall using mixed-precision - HPG-MxP on an exascale machine. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3712285.3759877

1 Introduction

Scientific computing has long relied on techniques from numerical linear algebra, nonlinear equations, numerical optimization, and ordinary and partial differential equations (PDEs) to model physical phenomena, make predictions, and explain them. Even in the age of artificial intelligence (AI), these continue to remain important [13].

Meanwhile, driven by the insatiable arithmetic compute throughput needs of large language models, vendors of high-performance computing (HPC) hardware are designing chips increasingly geared towards extremely high performance in dense matrix-matrix multiplication (GEMM) kernels in low precision formats. Graphics processing unit (GPU) vendors NVIDIA and AMD have designed 'tensor cores' and 'matrix units' respectively for a higher rate of growth in FP16, BF16 and INT8 GEMM throughput compared to IEEE FP32 and FP64 formats typically favored by scientific computing. Indeed, NVIDIA has championed support for FP8, FP6 and FP4 formats and claimed hundreds of petaflops on a single chip [1], unthinkable a few years ago. The energy usage per operation is also

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). The U.S. government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759877

lower when tested on matrix-matrix multiplication workloads. In fact, energy savings from mixing the use of lower precision formats has been shown in the past even for other non-AI workloads [5, 17].

However, since many scientific computing applications rely on higher precision, typically IEEE FP64, this throughput and efficiency is difficult to access for scientific computing. Computational scientists are thus starting to look into the use of lower precision formats in mixed-precision numerical algorithms [20].

Furthermore, many real-world scientific and engineering computing workloads are limited by memory bandwidth rather than arithmetic throughput [16]. These include a large number of simulation applications that rely on solving PDEs governing fluid mechanics, solid mechanics, heat transfer, electromagnetism, chemically-reacting flows and plasma physics using methods such as finite differences, finite volumes, finite elements and lattice Boltzmann. Such applications do not use tensor cores at all since they do not rely on GEMM as their main computational motif. The primary computational motifs in simulation codes tend to be from sparse linear algebra - sparse matrix vector products (SpMV), sparse triangular solves (SpTRSV), sparse matrix sparse matrix products (SpGEMM), and dot products (DOT). At a slightly higher level, multigrid methods play an important role in the scalable solution of PDEs, and present unique challenges in accelerated distributed HPC [6]. The US Exascale Computing Project invested in developing mixed precision numerical algorithms for some of these motifs [2].

The High Performance GMRES Mixed Precision (HPG-MxP) benchmark was proposed [31] to measure the performance of a supercomputer on such memory-bandwidth limited simulation workloads. In contrast to the existing High Performance Conjugate Gradient benchmark [14], it allows the use of mixed precision internally, while requiring a solution 'somewhat close' to that obtained by a fully double-precision solver (to be clarified later). In our view, the objective of the benchmark is to get a practical upper limit for the performance of mixed-precision memory-bandwidth limited workloads while achieving essentially the same usefulness as double-precision computation. The work of developing and running at scale an optimized benchmark achieving the maximum possible performance will serve several purposes:

(1) it will serve as a yardstick to shoot for while optimizing the performance of scientific simulations applications, especially ones utilizing implicit solvers that require the solution of large sparse linear systems,
(2) the learning from this activity will help guide computational scientists and HPC engineers on the best ways to utilize mixed-precision methods to accelerate workloads on their HPC systems, and
(3) it will guide hardware vendors and other library providers in designing and optimizing their numerical libraries to best support mixed-precision simulation workloads.

Our contributions in this paper are as follows.

(1) We describe a state-of-the-art implementation of the HPG-MxP benchmark that achieves much higher performance than the reference implementation on large-scale GPU-based HPC systems.
(2) We show that with such an optimized implementation, a higher speedup than earlier reported is possible from a double-single mixed-precision solver on current-generation exascale systems. While the benchmark allows for the use of any precision format in most steps of the solver algorithm, we focus on the use of single precision as the only low precision format for this paper.
(3) We report, for the first time, a full-system HPG-MxP run (9408 nodes) on the world's first exascale system, Frontier, at the Oak Ridge National Laboratory, USA, using our optimized codebase.
(4) We report some performance analysis of the benchmark code, particularly traces showing the achieved compute-communication overlap, the achieved memory bandwidth and performance relative to the roofline.
(5) We confirm that validation on a small fixed problem size (on a single node) is sufficient to accurately penalize our mixed precision solver. This is achieved by introducing a full-scale validation that uses all available nodes and the full problem size.

2 Background

The first major attempt to make system benchmarking better reflect real scientific workloads came with the introduction of the High Performance Conjugate Gradient (HPCG) benchmark [14]. Since HPG-MxP is based on HPCG, we first give an overview of HPCG. It solves the three-dimensional Poisson equation, a fundamental PDE from which most PDE theory and solver techniques derive, using a 27-point finite difference discretization on a uniform Cartesian mesh of a cube-shaped domain. This results in a matrix with as many rows as mesh points, and 27 non-zero values per row for interior points. Boundary points have fewer non-zeros depending on whether they lie on a face, edge or corner point. The precon-

Algorithm 1 Preconditioned conjugate gradient algorithm for a symmetric positive-definite linear system $\boldsymbol{Ax} = \boldsymbol{b}$

Require: Initial guess $\boldsymbol{x}_0$
  (preconditioner generation) $\boldsymbol{M} \leftarrow \mathcal{P}(\boldsymbol{A})$
  $\boldsymbol{p}_0 \leftarrow \boldsymbol{x}_0, \boldsymbol{r}_0 \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{p}_0$, $i \leftarrow 1$.
  while $i < N$ **do**
    (preconditioner application) $z_i \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}_{i-1}$
    if $i = 1$ **then**
      $\boldsymbol{p}_i \leftarrow \boldsymbol{z}_i$
      $\alpha_i \leftarrow (\boldsymbol{r}_{i-1}, \boldsymbol{z}_i)$
    else
      $\alpha_i \leftarrow (\boldsymbol{r}_{i-1}, \boldsymbol{z}_i)$
      $\beta_i \leftarrow \frac{\alpha_i}{\alpha_{i-1}}$
      $\boldsymbol{p}_i \leftarrow \beta_i \boldsymbol{p}_{i-1} + \boldsymbol{z}_i$
    end if
    $\alpha_i \leftarrow (\boldsymbol{r}_{i-1}, \boldsymbol{z}_i)/(\boldsymbol{p}_i, \boldsymbol{A}\boldsymbol{p}_i)$
    $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \alpha_i \boldsymbol{p}_i$
    $\boldsymbol{r}_i \leftarrow \boldsymbol{r}_{i-1} - \alpha_i \boldsymbol{A}\boldsymbol{p}_i$
  end while

ditioned Conjugate Gradient (CG) method is used as the solver. This is a Krylov subspace solver that is guaranteed to converge, for

symmetric positive-definite matrices, in N iterations where N is the dimension of the matrix. The version used by the benchmark [14] is shown in algorithm 1. In HPCG, the preconditioner is required to be one cycle of geometric multigrid. Multigrid is a family of highly scalable algorithms to solve a PDE-based problem [10], based on the ideas of error smoothing and coarse grid correction. Multigrid methods are based on discretizing the problem on a hierarchy of meshes of coarser resolutions and applying an iterative method at each mesh, or multigrid level, to 'smooth' the errors. A two-grid cycle is shown in figure 1, which is applied recursively to solve the coarse-grid problem $\boldsymbol{A}_H$ to generate a multigrid cycle.

In a well-tuned mathematically-sound implementation, for a stencil-based matrix of size $N \times N$, multigrid converges in $O(N)$ operations , with a small factor [10] ('textbook multigrid'). However, this assumes a fixed coarse grid problem size independent of N on the coarsest level, with the number of multigrid levels increasing to scale up to ever larger fine-grid problem sizes. Since HPCG fixes the number of multigrid levels to 4, this ideal scalability is not expected.

HPCG uses the symmetric Gauss-Seidel iteration as the smoother in its multigrid preconditioner. If the system matrix $\boldsymbol{A}$ is split into lower triangular $\boldsymbol{L}$, upper triangular $\boldsymbol{U}$ and diagonal parts $\boldsymbol{D}$, the iteration for $\boldsymbol{Az} = \boldsymbol{r}$ can be written as

$$(\boldsymbol{D} + \boldsymbol{L})\boldsymbol{y} = \boldsymbol{r} - \boldsymbol{U}\boldsymbol{z}, \tag{1}$$

$$(\boldsymbol{D} + \boldsymbol{U})\boldsymbol{z}^{(1)} = \boldsymbol{r} - \boldsymbol{L}\boldsymbol{y}, \tag{2}$$

where $\boldsymbol{y}$ is a temporary intermediate vector. Note that this involves a lower SpTRSV and SpMV with an upper triangular matrix, followed by an upper SpTRSV and SpMV with a lower triangular matrix. In an efficient implementation, the whole symmetric Gauss Seidel iteration can be done in two kernels.

There are two traditional approaches to finding parallelism in the otherwise sequential Gauss-Seidel iteration - level scheduling and multicoloring [19, section 2.7.1]. While a level-scheduled triangular solve preserves the original ordering of the matrix but only has a limited amount of parallelism, multicoloring methods use an independent set ordering to find independent sets expose more parallelism. Typically, this means level-scheduled methods deliver the same preconditioning effectiveness to the CG solver though cannot utilize the GPU very effectively, while multicolored triangular solves may degrade the preconditioner quality somewhat (decreasing CG's convergence rate) but delivers good GPU utilization.

To map the problem to a distributed parallel computer, the framework of domain decomposition is used [27]. The spatial domain under consideration, which is discretized by a mesh or graph, is partitioned among the processors. Each mesh cell or graph point is associated with a row of the matrix, and the nonzeros in that row denote its neighborhood (which may not be the same as the topological neighborhood in the mesh). Thus, the matrix is distributed by row - each processor owns a block of rows and all columns of the matrix.

In HPCG, the processors are factored into a 3D grid, similar to the mesh itself. Thus a grid of size $N_x \times N_y \times N_z$ is uniformly divided amongst $p_x \times p_y \times p_z$ processors. Assuming an isotropic grid that is a perfect cube both in points and processors, N^3 points are mapped to p^3 processors. Each processor has $(\frac{N}{p})^3$ points . Let $n := N/p$. In a typical finite difference or finite volume discretization, each mesh point or cell directly interacts with $O(1)$ neighbors, that is, a constant number of neighbors independent of the problem size. Thus, in a regular Cartesian mesh, each row has a fixed constant number of nonzeros, except for those corresponding to global boundary points. Therefore, operations such as SpMV and Gauss-Seidel iterations compute on all the points in the subdomain owned by a processor, and need to communicate with the processors owning their nearest spatial neighboring subdomains. For cubic subdomains as in HPCG, one requires $O(n^3)$ operations of computations, and about $6n^2$ volume of communications with neighboring processors. Seen another way, if ν is the problem size per processor, local compute scales as $O(\nu)$ while communication volume scales as $O(\nu^{2/3})$. Thus, the communication volume is a geometric order lower than the local computation volume, and therefore, network bandwidth is typically not the limiting factor for HPCG performance. In most cases, HPCG is limited by local memory bandwidth due to the low constant arithmetic intensity of the computations.

HPCG is a good measure of a HPC system's performance on real workloads that are limited by memory bandwidth. However, many real-world problems involve nonsymmetric matrices, to which the CG algorithm is not applicable. In such cases, the Generalized Minimum Residual (GMRES) solver is popular. It is also a Krylov Subspace solver that, as the name implies, tries to minimize the 2-norm of the residual of the linear system in each iteration [26]. However, unlike in the symmetric case, there is no 'short-term' recurrence formula, which means previous iterations' Krylov basis vectors need to be stored. This significantly increases the memory requirement. Certain variants, such as using CGS2 reorthogonalization (see section 3), also use some dense BLAS2 routines.

Furthermore, HPCG is required to use double-precision arithmetic. As stated in the introduction, it is of interest to find mixed-precision algorithms for PDE-based and other simulation workloads, and it is thus of interest to measure HPC systems' expected performance on such workloads. Research in mixed-precision methods for sparse matrices is not new. In 2008, Buttari et al. [11] investigated mixed-precision CG and GMRES solvers, among other mixed-precision solvers for sparse problems. More recently, Loe et al. [21] implemented mixed-precision GMRES using two different strategies - starting a single-precision solver and then switching to double after some iterations, and iterative refinement (GMRES-IR), described in the next section. They used either polynomial preconditioning or block-Jacobi preconditioning, whose characteristics in terms of preconditioning effectiveness, parallelism and resource utilization are quite different from the multigrid preconditioner prescribed by HPG-MxP. They test their implementation using sample sparse matrices from applications on a single NVIDIA V100 GPU.

HPG-MxP, then known as the High Performance GMRES Multiprecision (HPGMP) benchmark, was first proposed in 2022 [31]. It solves a similar problem as the HPCG benchmark, but as the name suggests, uses a GMRES solver and allows the use of lower precisions.

As far as the authors are aware, Anzt et al. [3] were the first to run HPG-MxP on the Frontier exascale system at Oak Ridge National Laboratory.

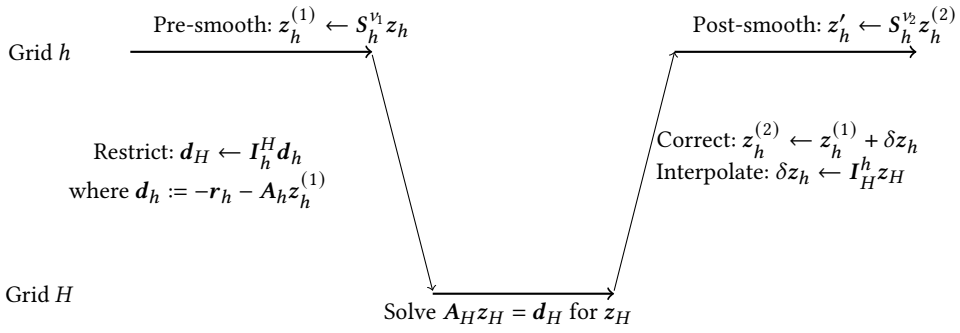


Figure 1: Cycle of the linear two-grid method for the system $A_h z_h = r_h$. z_h is an initial guess or prior approximation of the solution on the fine grid, S is the smoothing iteration or smoother, I_h^H denotes the restriction operator, I_H^h denotes the prolongation or interpolation operator, and z'_h is the improved solution approximation.

3 The HPG-MxP benchmark

HPG-MxP, introduced as HPGMP by Yamazaki et al. [31] in 2022, is the first benchmark to target mixed-precision simulation workloads. Similar to HPCG, it solves a Poisson-like problem on a uniform mesh. This time, there is an option to make the problem nonsymmetric, though as Yamazaki et al. observe, for GMRES the symmetric version is at least as difficult to solve as the nonsymmetric one. The matrix they construct is weakly diagonally dominant; that is, for each row i, $\sum_{j \neq i} |a_{ij}| \leq a_{ii}$.

The original right-preconditioned GMRES algorithm [25, chapter 9] is shown in algorithm 2. It attempts to minimize the 2-norm of the residual, and generation of orthogonal Krylov basis vectors is necessary for this. The Arnoldi process used to build the orthogonal Kyrlov basis may use one of a few different methods, such as classical Gram-Schmidt and modified Gram-Schmidt. While classical Gram Schmidt is amenable to more efficient implementation, it is more prone to round-off errors and resulting loss of orthogonality.

Algorithm 2 is called 'right-preconditioned' since the preconditioner applies on the right - it is an algorithm to solve the linear system $\boldsymbol{A}\boldsymbol{M}^{-1}y = \boldsymbol{b}$, whose solution $\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{y}$ is the same as that for $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. The preconditioner is one cycle of geometric multigrid with a forward Gauss-Seidel smoother. The restriction $\boldsymbol{R}$ is a simple injection from every alternative fine grid point, while the prolongation operator is its transpose $\boldsymbol{P} = \boldsymbol{R}^T$. If $c_f(i)$ is the index of the ith coarse grid point in the fine grid and $\boldsymbol{v}$ is a fine grid vector,

$$(\boldsymbol{R}\boldsymbol{v})_i := v_{c_f(i)}. \tag{3}$$

In the HPG-MxP benchmark, mixed-precision is utilized at the highest level via the idea of iterative refinement applied to GMRES, GMRES-IR. Since GMRES attempts to generate orthogonal Krylov basis vectors, loss of orthogonality is detrimental to its convergence [15]. This is especially a problem with classical Gram-Schmidt orthogonalization and the usage of lower precisions. Therefore, the version of GMRES-IR prescribed by the benchmark uses reorthogonalization steps to better preserve orthogonality.

We show the GMRES-IR CGS2 algorithm 3 used by the benchmark, including details of how the least-squares problem (QR factorization) is solved using Given's rotations. The key aspect to note here is that many of the operations are allowed to be performed in low precision, but the residual update in line 7 and solution update in line 47 are required to be done in double-precision. This makes it possible for the solution to converge to an accuracy level equivalent to that of a fully double precision solver. Note that the QR factorization update using Given's rotations happens on the CPU, on each process redundantly.

Algorithm 2 Right-preconditioned GMRES algorithm for a non-singular linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$

Require: Initial guess $\boldsymbol{x}_0$
  $\boldsymbol{M} \leftarrow \mathcal{P}(\boldsymbol{A})$ ▷ Preconditioner generation
  $\boldsymbol{r}_0 \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\beta = ||\boldsymbol{r}_0||$, $\boldsymbol{q}_1 = \boldsymbol{r}_0/\beta$.
  for $j = 1, 2, ..., m$ do **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{q}_j$ ▷ Preconditioner application and SpMV
    for $i = 1, 2, 3, ..., j$ **do**
      $h_{i,j} \leftarrow (\boldsymbol{w}, \boldsymbol{q}_i)$
      $\boldsymbol{w} \leftarrow \boldsymbol{w} - h_{i,j}\boldsymbol{q}_i$
    end for
    $h_{j+1,j} \leftarrow \|\boldsymbol{w}\|_2$, $\boldsymbol{v}_{j+1} \leftarrow \boldsymbol{w}/h_{j+1,j}$
    $V_m := [\boldsymbol{v}_1, ..., \boldsymbol{v}_m]$, $\bar{\boldsymbol{H}}_m \leftarrow \{h_{i,j}\}_{1 \leq i \leq j; 1 \leq j \leq m}$
  end for
  $\boldsymbol{y}_m \leftarrow \text{argmin}_y \|\beta \boldsymbol{e}_1 - \bar{\boldsymbol{H}}_m \boldsymbol{y}\|_2$
  $\boldsymbol{x}_m \leftarrow \boldsymbol{x}_0 + \boldsymbol{M}^{-1}\boldsymbol{Q}_m \boldsymbol{y}_m$
  if $||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m||_2 < \tau$ **then**
    Exit.
  else
    Restart with $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}_m$.
  end if

Though they provide the option of using a nonsymmetric matrix for the benchmark, Yamazaki et al. [31] prefer the same symmetric weakly diagonal-dominant matrix as in HPCG, since this matrix actually takes more iterations for GMRES to converge than their nonsymmetric variant. The matrix has all diagonal entries equal to 26 and all off-diagonal entries equal to -1.

The benchmark consists of three phases:

(1) validation,
(2) mixed-precision benchmark, and
(3) double-precision 'reference' benchmark.

Algorithm 3 Right-preconditioned mixed-precision GMRES-IR for the linear system $\boldsymbol{Ax} = \boldsymbol{b}$. Steps in blue are allowed to be performed in low or mixed precision.

Require: Initial guess $\boldsymbol{x}_0$, restart length m, tolerance τ.

$\boldsymbol{M} \leftarrow \mathcal{P}_{mg}(\boldsymbol{A})$ ▷ Multigrid preconditioner generation
$\rho_0 \leftarrow \|\boldsymbol{b}\|_2$
$\boldsymbol{t} \in \mathbb{R}^{m+1}, \boldsymbol{c} \in \mathbb{R}^{m+1}, \boldsymbol{s} \in \mathbb{R}^{m+1}$
$\boldsymbol{H} \in \mathbb{R}^{(m+1)\times m}$ ▷ Projected system matrix
$\boldsymbol{Q} = [\boldsymbol{q}_1, \boldsymbol{q}_2, ..., \boldsymbol{q}_m] \in \mathbb{R}^{n\times m}$ ▷ Krylov basis vectors
while not converged **do**
  $\boldsymbol{r} \leftarrow \boldsymbol{b} - \boldsymbol{Ax}$, $\rho = ||\boldsymbol{r}||_2$
  if $\rho/\rho_0 < \tau$ **then**
    Converged, break.
  end if
  $\boldsymbol{r} \leftarrow \boldsymbol{r}/\rho$
  $\boldsymbol{q}_1 \leftarrow \boldsymbol{r}$
  $t_{0,0} \leftarrow \rho$.
  for $k = 1, 2, ..., m$ do **do**
    if $\rho/\rho_0 < \tau$ **then**
      Converged, break.
    end if
    $\boldsymbol{z} \leftarrow \boldsymbol{M}^{-1}\boldsymbol{q}_k$ ▷ Multigrid preconditioner
    $\boldsymbol{q}_{k+1} \leftarrow \boldsymbol{Az}$ ▷ SpMV to get next basis vector
    procedure CGS2 ORTHOGONALIZATION($\boldsymbol{q}_{k+1}$)
      $\boldsymbol{h} \leftarrow \boldsymbol{Q}^T_{[1:k]}\boldsymbol{q}_{k+1} \in \mathbb{R}^k$ ▷ GEMVT
      $\boldsymbol{q}_{k+1} \leftarrow \boldsymbol{q}_{k+1} - \boldsymbol{Q}_{[1:k]}\boldsymbol{h}$ ▷ GEMV
      $H_{1:k,k} \leftarrow \boldsymbol{h}$
      $\boldsymbol{h} \leftarrow \boldsymbol{Q}^T_{[1:k]}\boldsymbol{q}_{k+1}$ ▷ reorth. GEMVT
      $\boldsymbol{q}_{k+1} \leftarrow \boldsymbol{q}_{k+1} - \boldsymbol{Q}_{[1:k]}\boldsymbol{h}$ ▷ reorth. GEMV
      $H_{1:k,k} \leftarrow H_{1:k,k} + \boldsymbol{h}$
    end procedure
    $\beta \leftarrow \|\boldsymbol{q}_{k+1}\|_2$
    $\boldsymbol{q}_{k+1} \leftarrow \boldsymbol{q}_{k+}/\beta$ ▷ Normalize the new basis vector
    $H_{k+1,k} \leftarrow \beta$
    procedure UPDATE QR WITH GIVEN'S ROTATIONS
      for $j = 1, 2, ..., k-1$ **do**
        $H_{j+1,k} \leftarrow -s_j H_{j,k} + c_j H_{j+1,k}$
        $H_{j,k} \leftarrow c_j H_{j,k} + s_j H_{j+1,k}$
      end for
      $\mu \leftarrow \sqrt{H^2_{k,k} + H^2_{k+1,k}}$
      $H_{k,k} \leftarrow \mu$
      $H_{k+1,k} \leftarrow 0$
      $\rho \leftarrow |t_k s_j|$.
      $t_{k+1} \leftarrow -t_k s_j$
      $t_k \leftarrow t_k c_j$
      $s_k \leftarrow s_j$, $c_k \leftarrow c_j$
    end procedure
  end for ▷ Restart cycle completed
  $\boldsymbol{t} \leftarrow \boldsymbol{H}^{-1}\boldsymbol{t}$ ▷ Dense TRSM of size m
  $\boldsymbol{r} \leftarrow \boldsymbol{Qt}$
  $\boldsymbol{x}_m \leftarrow \boldsymbol{x}_0 + \boldsymbol{M}^{-1}\boldsymbol{r}$ ▷ Mixed-precision update
end while

The benchmark first validates the mixed-precision solver on a few, fixed number of processors. By default, this is all the GPUs on one node. The validation consists of first running the double precision GMRES solver, starting from a zero initial guess, to converge the residual norm by 9 orders of magnitude. The number of iterations required, n_d, is recorded. Next, mixed-precision GMRES-IR is run to converge to the same tolerance, starting from a zero initial guess again. The number of iterations required, n_{ir}, is recorded.

This is followed by the benchmark phase, where mixed-precision GMRES-IR is first run for a fixed number of iterations. The time taken by each motif is recorded and the number of floating point operations is counted using a carefully constructed model. Floating point operations of different precisions are counted equally, and thus the reported GFLOP/s number is a mixed-precision number, not the standard double-precision GFLOP/s. The final GFLOP/s metric is penalized by the ratio of the iteration counts obtained in the validation phase. That is, the final floating point throughput F is given by $F = F_{\text{raw}}\frac{n_d}{n_{ir}}$. Thus, to avoid a heavy penalty, mixed-precision GMRES-IR must not take too many additional iterations to converge 9 orders of magnitude. It is in this sense that the mixed-precision solver is required to provide a solution that is 'somewhat close' to that of the double-precision solver. When the ratio $\frac{n_d}{n_{ir}}$ is less than 1, it is multiplied by the final GFLOPS value to penalize the mixed precision run. However, then the ratio is greater than 1, no penalty is applied. Thus, in the event that the mixed precision solver actually takes *fewer* iterations to converge for any reason, this is not regarded as an advantage for the mixed precision solver, and it is regarded as though the mixed precision solver has the same convergence rate as double precision GMRES.

The GMRES-IR solution process is repeated, starting from a zero initial guess each time, until the requested running time is filled. Similar to HPCG, the official running time proposed by Yamazaki et al. [31] is 1800 seconds. Results from this phase are labelled as 'mxp' in the results section.

Finally, a double-precision GMRES solver is run for the same number of fixed maximum iterations, and similar performance metrics are collected. We report results from this phase labelled as the 'double' in the results section.

3.1 The reference implementation

The reference implementation used in this work is from the official HPG-MxP repository [1] as of April 2025. As described by Yamazaki et al. [31], there are many substantial inefficiencies in this version. This is understandable since their aim was to propose the benchmark, not provide a fully optimized implementation. However, we aim for an optimized implementation, for which we document the issues with the reference implementation first.

(1) The Gauss-Seidel implementation uses SpTRSV from cuSparse and rocsparse, which use a level-scheduled implementation [23] without reordering. This variant is mathematically equivalent to a sequential lexicographic (ordered spatially by mesh points) Gauss-Seidel, but it does not expose the most parallelism and does not fully utilize the GPU [28].

[1] https://github.com/hpg-mxp/hpg-mxp

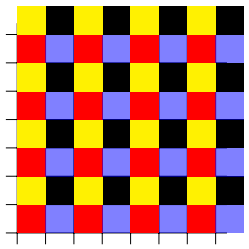
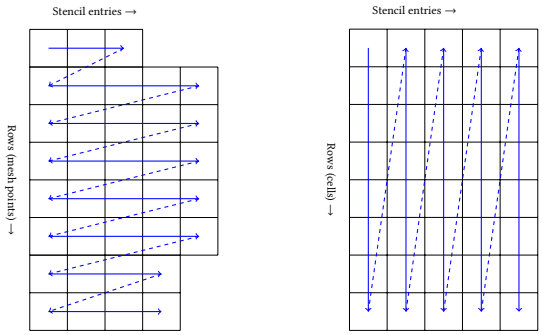


Figure 2: An example of independent sets for a 2D 9-point stencil, the 2D analog of the 3D 27-point stencil used in HPCG and HPG-MxP



(a) CSR **(b) ELL**

Figure 3: Two possible layouts of the non-zero coefficients' array of a matrix

(2) Forward Gauss-Seidel is performed using a separate SpMV with the $\boldsymbol{U}$ matrix followed by a SpTRSV with the $\boldsymbol{L}$ matrix, which is wasteful.
(3) In multigrid, following the Gauss-Seidel smoother, the smoothed fine-grid residual is first computed using SpMV, followed by restriction of this to the coarse grid. This performed substantial additional work which is not necessary, since the restriction is simple injection of values from selected fine-grid points to the coarse grid.
(4) There is no overlap of communication and computation in either the SpMV or the Gauss-Seidel operation. Indeed, the code does not support any asynchronous behaviour.
(5) The sparse matrix format used is compressed sparse row, or CSR. While this format is a popular choice and reasonably efficient in general, on GPU architectures it has been shown that for stencil-based problems like the one here, other formats can work better [8].
(6) All mixed-precision operations are done on the host, necessitating additional host-device copies and utilizing slower CPU DRAM.

3.2 Optimizations

In response to the issues identified with the reference implementation, we carry out several algorithmic changes and optimizations.

3.2.1 Multicolor Gauss-Seidel iteration. To start with, we implement the forward Gauss-Seidel in its 'relaxation' version [19], completing the operation in one sweep over the matrix.

More importantly, we reorder the matrix and vectors symmetrically using an independent set ordering in order to expose fine-grained parallel work in the Gauss-Seidel kernel on GPUs. Each subdomain is reordered independently, without any communication. If the n local mesh points can be divided into n_c independent sets such that no two points within a set are directly connected to each other in the sparsity pattern of the system matrix, a Gauss-Seidel iteration can be completed in n_c operations one after the other, each working on n/n_c rows in a fully parallel manner. If $n_c = O(1)$ independent of n, we should be able to achieve good parallel efficiency on a GPU.

The ordering itself is computed on the GPUs using the Jones-Plassmann-Luby (JPL) algorithm [18, 22] in the optimization function of the benchmark. Naumov et al. introduced a GPU implementation [24]. We use the implementation by Trost et al. [29].

As shown in figure 2, the JPL algorithm as well as a sequential greedy algorithm [25, section 3.3.3] applied to a 9-point stencil on a spatially two-dimensional (2D) mesh give 4 independent sets, or 'colors'. The analogous 27-point stencil in 3D requires 8 colors. This enables the use of multicolored Gauss-Seidel, which involves fully parallel operations on the rows within each color. The convergence rate sometimes suffers compared to lexicographic Gauss-Seidel and other types of reorderings such as Reverse Cuthill McKee (RCM) [9, 19]. However, this is less of an issue within a multigrid preconditioner setting.

3.2.2 ELLPACK matrix format. We use the ELLPACK matrix format [8], sometimes simply referred to as ELL. This format is able to fully utilize GPU warps, especially the 64-wide warps of AMD GPUs, when there are only a few non-zeros per row. Fully utilizing the warps is more difficult using the CSR format [19, section 2.9], which is used in the reference implementation of HPG-MxP by Yamazaki et al. [31]. The memory layouts used by the two formats are depicted in figure 3. While the ELL format may have some overhead in its values and column-indices array due to padding in rows having less than the maximum number of nonzeros, it does not require a row pointer array.

3.2.3 Compute-communication overlap. Similar to rocHPCG [29], we implement SpMV and Gauss-Seidel operations that update the interior points while neighborhood halo communication operations take place asynchronously. Pre-conditions for each operation are ensured using two GPU streams ('compute' stream and 'halo' stream) and one event per MPI rank. The non-blocking kernels that compute the matrix-vector product for the both the interior+boundary and halo regions are enqueued on the 'compute' stream, while the non-blocking buffer packing kernel and asynchronous host-device copies (if used) are enqueued on the 'halo' stream.

The event is the object that achieves precise synchronization between the two streams during the Gauss-Seidel operation. This operation has a significant difference from the SpMV operation - unlike in SpMV, Gauss-Seidel (equation (1)) requires the output vector $\boldsymbol{y}$ to be communicated, not the input vector. The event is used to ensure that the interior computation kernel begins only after boundary entries of the initial vector $\boldsymbol{y}$ have been copied into the send buffer, since the interior kernel updates boundary entries in addition to fully interior locations.

3.2.4 Fused SpMV-restriction. The original implementation of Yamazaki et al. [31] explicitly stores the restriction matrix $\boldsymbol{R}$. Representing the fine grid index set as F and the coarse grid by C, the coarse grid residual is computed as

$$v_i \leftarrow \sum_{j \in \mathcal{N}_i} A^f_{ij} x^f_j, \quad \forall\, i \in F \tag{4}$$

$$r^c_i \leftarrow \sum_{j \in \mathcal{N}_i} R_{ij}(b_j - v_j) \quad \forall\, i \in C. \tag{5}$$

where $\boldsymbol{b}$ is the right-hand side vector and $\boldsymbol{x}^f$ is the pre-smoothed solution vector on the fine grid. Since the restriction is a simple injection as shown in equation (3), we can fuse the residual calculation and restriction operations. Thus, in our implementation, we compute the smoothed residual only at the coarse grid points, rather than separately compute it at all fine grid points and then restrict to the coarse grid. Using the coarse-to-fine grid index mapping f_c,

$$r^c_i \leftarrow b_{f_c(i)} - \sum_{j \in \mathcal{N}(f_c(i))} A^f_{f_c(i)j} x^f_j \quad \forall\, i \in C. \tag{6}$$

Note that we do not store the restriction operator explicitly. We updated the accounting of the number of floating point operations in the multigrid preconditioner to include this optimization.

3.2.5 Software engineering. Given that the reference implementation is cross-platform and runs on both AMD and NVIDIA GPUs, we preserved this aspect and made sure that our implementation works with high performance on both vendors' devices. Taking inspiration from numerical libraries like Ginkgo [4], we introduce a device context `DeviceCtx` that abstracts many vendor-specific details, including device memory allocation and deallocation, GPU stream and event operations, initialization and destruction of BLAS and sparse BLAS library handles etc. We use C++ features like function overloading and templates to generate high-performance kernels for both CUDA and AMD devices. For example, in sparse matrix-vector product, values loaded from the input vector are constant and may be used more than once, and it makes sense to cache them in L1 cache. However, the matrix nonzeros are read only once, it make sense to skip temporal caching for these values and improve performance slightly. The intrinsic (backend-specific, non-standard) functions for these types of loads and stores are different for the two platforms and we take this into account using C++ features.

Note that all of the improvements detailed so far apply to both the mixed-precision and purely double-precision solvers.

Additionally, we implement simple custom mixed-precision GPU kernels for operations such as WAXPBY as required by the mixed-precision GMRES-IR implementation. This allows us to remove the host-device copies performed by the reference implementation to perform these operations on the CPU.

3.3 Alternative full-scale validation

Yamazaki et al. argue [31] that validating the mixed-precision GMRES-IR on a small number of processes (typically 1 node) and corresponding small problem size is sufficient. The reason given is that the benchmark's purpose is to measure the computer's ability to perform operations representative of typical HPC applications while allowing the use of different precision formats, not to provide a truly scalable solver. However, as they themselves note, when the multigrid preconditioner is used, the loss of convergence rate when using mixed precision GMRES-IR can be worse at larger scales. In order to investigate the impact of validation at a 1-node scale on the penalty factor, we introduce a new validation mode in our version of the benchmark code.

In the new validation mode, all the processes available to the run and used for the benchmarking phase, are also used for the validation phase. The global problem size used for the two phases is also the same. Two modes are provided:

(1) `standard`: Double-precision GMRES is run on a small subset of processes, 1 node, until a relative residual norm of 10^{-9} is reached. Since the problem size is correspondingly small, this always happens within the iteration limit of 10,000. Mixed-precision GMRES-IR is then also converged 9 orders of magnitude and the number of iterations n_{ir} is recorded.
(2) `fullscale`: Double precision GMRES is run for a maximum of n_d (10,000) iterations or a relative residual norm of 10^{-9}, whichever comes first. The achieved relative residual norm τ is recorded. Mixed precision GMRES-IR is then run and converged until the same relative residual norm τ is achieved, and the number of iterations n_{ir} is recorded.

As Yamazaki et al. [31] noted, GMRES takes more and more iterations to converge to a fixed tolerance as the problem scale increases. Thus, with our new validation path, at low scales, the GMRES solver hits the 10^{-9} tolerance much before it reaches 10,000 iterations. The mixed precision GMRES-IR is then required to converge to 10^{-9}. However, at large scales, the global problem size is much larger and GMRES hits 10,000 iterations first. Eg., at 1024 nodes in our runs, the solver achieves a relative residual of about 1.15×10^{-5}. This was chosen in order to cap the amount of time the whole benchmark takes at very large scales, while still learning something about any loss of convergence caused by the use of mixed precision.

4 Results

We ran the optimized code on the Frontier system at Oak Ridge National Laboratory using AMD ROCm 6.2.4, Cray MPICH 8.1.31 and GCC 14.2. Each node consists of a 64-core AMD Milan CPU and 4 AMD MI250x GPUs, each divided into two Graphics Compute Dies (GCDs). Each GCD is effectively treated as a separate GPU. Thus, we consider there to be 8 GPUs per node. Each GCD is equipped with 64 GB of High Bandwidth Memory (HBM) with a vendor-claimed peak bandwidth of 1.6 TB/s. This HBM is generally referred to as the 'global' memory of the GPU device, as opposed to its much smaller but faster L2 and L1 caches. The CPU portions of the code (problem generation etc.) utilize OpenMP parallelism as in the reference code.

Table 1 summarizes the parameters we used to run the benchmark. We use a restart length of 30, similar to Yamazaki et al. [31]. This is also the default restart length in the popular PETSc package [7].

Anzt et al. [3] ran the reference version of the code on Frontier from 1 to 8192 nodes, using the reference implementation of Yamazaki et al. Due to the inefficiencies in this implementation detailed earlier, we do not expect it to give the best performance. However, as of writing, it is the state of the art in HPG-MxP performance on Frontier, so we include it in our results. Please note that

Parameter	Value
Restart length	30
Local mesh size	320^3
Specified running time (< 1024 nodes)	1800 s
Specified running time (>= 1024 nodes)	900 s
Max. GMRES iterations per solve	300
No. GCDs used for validation	8
Relative convergence tolerance for validation	1e-9

Table 1: HPG-MxP parameters used

in the graphs that follow, the points corresponding to the results by Anzt et al. [3], labelled "xsdk", are somewhat approximate as they have been read off a graph (figure 4 under the chapter 'Advances in mixed precision algorithms').

During validation on 1 node (8 GCDs), the reference double-precision GMRES solver takes 2305 iterations to converge 9 orders of magnitude, while the mixed-precision GMRES-IR requires 2382 iterations to converge to the same tolerance. This small increase in the required number of iterations is expected, and the appropriate penalty is applied to the mixed-precision performance metric.

4.1 Scaling, speedup and full-system performance

We first discuss the scaling results of the benchmarking phase. Figure 4 shows how the overall performance per GCD scales as we increase the problem size and the number of GCDs in proportion, for both our implementation ('present') and the reference implementation ('xsdk'). This is similar to weak scaling, though we do not regard this as true weak scaling because the solver does not converge to a specified residual tolerance, but rather executes a fixed number of iterations. We see that the performance holds up well up to large scales. However, as we approach the full system scale, the scaling efficiency decreases due to the many inner products required by the GMRES algorithm. Each inner product requires a global all-reduce operation. Even though the CGS2 version batches the inner product into a transposed GEMV operation and thus reduces the effective latency, we still see some degradation of the overall scaling. Depending on the mapping of subdomains to MPI ranks, the coarse multigrid levels may also contribute to some of this decrease in efficiency (see the discussion on tracing below and figure 9). Since the reference implementation achieves much lower performance in general, it does not see this effect. The weak scaling efficiency of our implementation from 1 node to 9408 nodes is 78%.

At the full system scale of 9408 nodes or 75,264 GPUs, we achieve an overall mixed-precision performance of 17.23 petaflops. For perspective, when we ran HPCG ourselves on Frontier on 9408 nodes, we achieved 10.4 petaflops. We note that these numbers are not directly comparable since the solvers are different in the two benchmarks.

Figure 5 shows the (penalized) speedup obtained by mixed single-double precision GMRES-IR versus double precision GMRES. We see a remarkable overall speedup of about 1.6×, against a theoretical peak of 2× for going fully to single precision assuming the code is limited by memory bandwidth. This is much improved compared to the speedups obtained using the reference implementation. Interestingly, about 1.6× was also the speedup reported by Buttari et

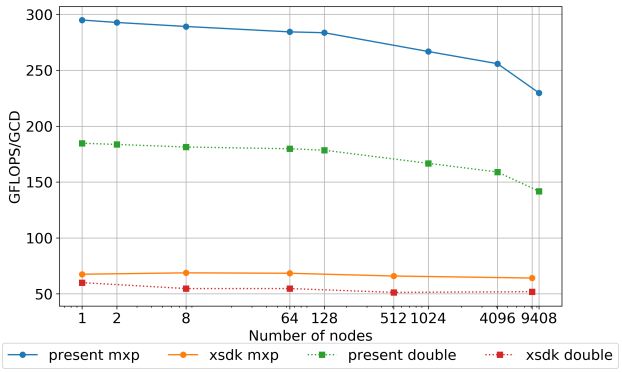


Figure 4: 'Weak' scaling of the overall benchmark on Frontier. It is based on the penalized time taken by the mixed-precision solver to complete the specified number of iterations.

al. [11] for mixed double-single precision GMRES on a single Intel Woodcrest CPU from the year 2006.

Clearly, the perfect speedup of the orthogonalization phase plays a role in this. Since this operation is a dense BLAS-2 operation, it makes the best use of increased memory throughput of lower precision numbers. At very large scales, however, the orthogonalization spends more time in MPI all-reduce operations, thus reducing the speedup somewhat. Multigrid (primarily Gauss-Seidel, as we shall see) and SpMV drag the speedup down somewhat owing to their the need to fetch index arrays, leading to lower arithmetic intensity and lower advantage from decreasing the bit-width of the floating point numbers. We note that optimizing the motifs in multigrid and SpMV as detailed in subsection 3.2 significantly improves the attained speedup.

Because of our cross-platform implementation discussed in section 3.2.5, we are able to seamlessly build and run on systems with NVIDIA GPUs. In passing, we observed (figure 6) similar speedups on a small commodity cluster containing NVIDIA K80 GPUs. The absence of tensor cores on these legacy GPUs is unlikely to affect the results, since the benchmark does not include any dense matrix-matrix operations on the GPU.

4.2 Validation methodologies

We compare the validation method of Yamazaki et al. [31] to the new validation method described in section 3.3. Recall that the validation phase computes the ratio of iteration counts $\frac{n_d}{n_{ir}}$ to penalize the performance of mixed-precision GMRES-IR to include the effects of any slowdown in convergence rate. The ratios computed by `standard` validation of Yamazaki et al. and the `fullscale` validation are shown in table 2. As explained in section 3, if the ratio is less than 1, it is considered a penalty for the mixed precision GFLOPS number. It turns out that the `standard` small-scale validation method is more or less as stringent as `fullscale` with 10,000 iterations. It is clear from the full-scale residual norms that, up through 8 nodes, the validation double precision solve hits the residual reduction criterion of 1e-9. However, once we get to a scale of 64 nodes, it hits the iteration limit of 10,000 iterations first, and does not reach 1e-9 residual reduction.

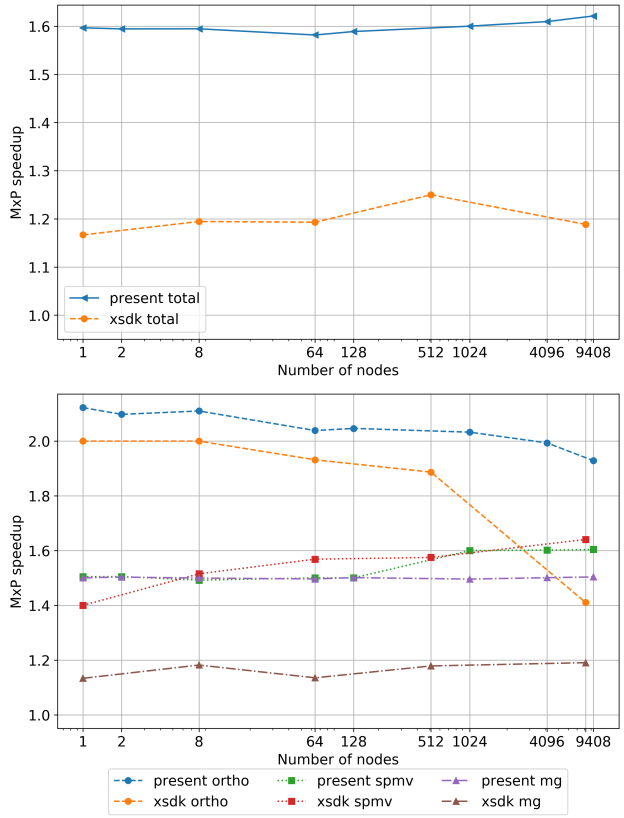


Figure 5: Speedups obtained on Frontier for different computational motifs in mixed-precision GMRES-IR over double precision GMRES. They are based on the penalized GFLOP/s rating of the mixed-precision solver to complete the specified number of iterations compared to that of the double-precision solver.

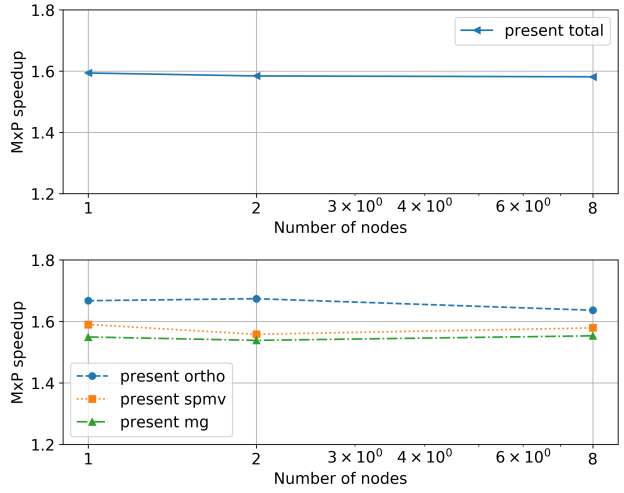


Figure 6: Speedups obtained on a small cluster with NVIDIA Tesla K80 GPUs

Nodes	Std ratio	Full-scale ratio	Full-scale relative residual norm
2	0.968	0.966	9.98e-10
8	0.968	1.008	9.99e-10
64	0.968	1.050	1.65e-6
128	0.968	1.023	2.82e-6
1024	0.968	1.067	1.154e-5
4096	0.968	0.958	1.148e-5

Table 2: Iteration ratios $\frac{n_d}{n_{ir}}$ for the two validation methods

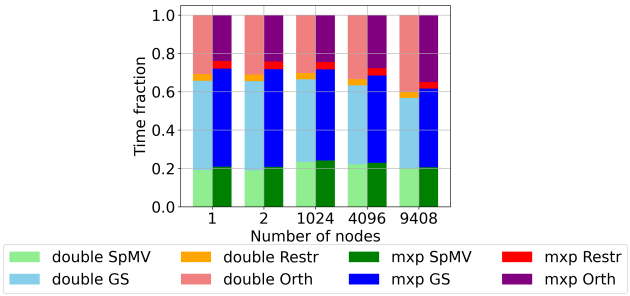


Figure 7: Breakdown of time spent in the multigrid smoother (GS), CGS2 orthogonalization (Ortho), sparse matrix vector product (SpMV) and multigrid restriction (Restr) on Frontier

4.3 Performance analysis

We take a look at the breakdown of the time spent in the different motifs in the benchmark at two different scales in Figure 7. The bar chart shows the four main motifs that take nearly all the time during the mixed-precision run and the reference double-precision run of the benchmark. As expected, the mixed-precision variant spends less time in orthogonalization, since this operation gets the most benefit from switching to single precision. Going from 1 node to 9408 nodes, the full system scale, we notice that the orthogonalization takes a greater share of time, likely because the all-reduce operations in the inner product operations require more time to synchronize.

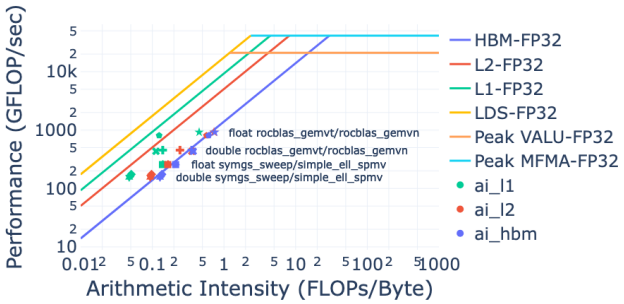


Figure 8: Roofline of the benchmark on a single GCD of an AMD MI250x. The ten most expensive kernels are depicted, eight of which are labeled. The unlabelled kernels are the double and single precision Fused SpMV-restriction, which perform similar to the Gauss-Seidel sweeps.

Figure 8 is a roofline of the most expensive kernels in the benchmark (both the optimized and reference version) on a single GCD, obtained from AMD's rocprofiler-compute. We see that the GFLOP/s obtained for the kernels lines up at the HBM bandwidth limit. That is, despite some utilization of L2 and L1 caches, the measured throughput is at the same level as the HBM limit. Thus, the kernels are memory bandwidth-limited, as expected.

The compute-communication overlap achieved by our implementation can be seen on the `rocprof` traces of a 'middle' rank that communicates with the maximum number of neighbors during an 8-node benchmark run on Frontier in figure 9. On the fine grid (figure 9a), we can see that host-device copy after halo buffer packing, as well as the actual communications, are completely hidden by the interior Gauss-Seidel kernel on the first independent set color. However, on the coarsest level (figure 9b), only the first independent set is not sufficient to completely overlap the communication. This is because the communication surface is larger here as a ratio of the computation volume, compared to the fine grid. Overlapping more of the Gauss-Seidel kernel with communication is possible and will be addressed in future work. For other operations like SpMV, the halo communications are effectively hidden by interior computations on all multigrid levels.

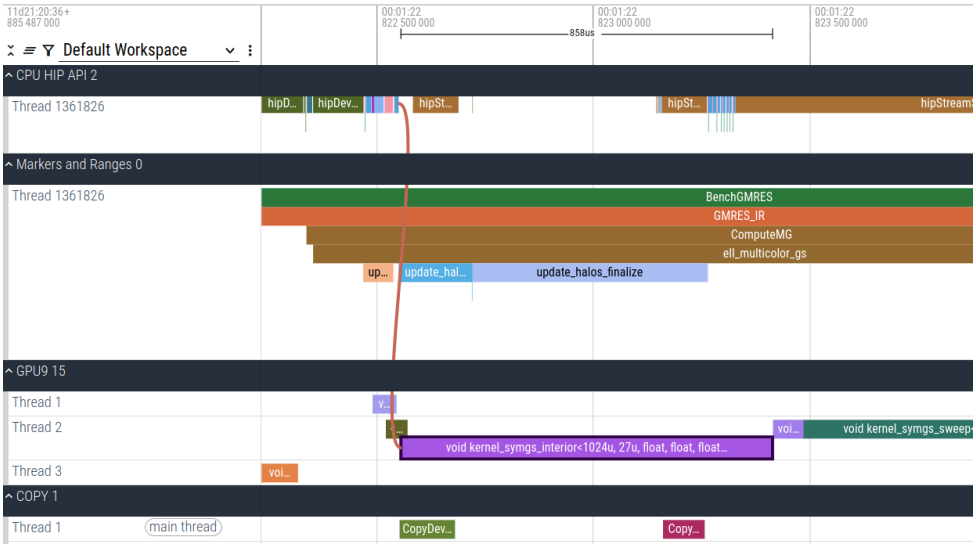
5 Conclusion

Our results show that simulation workloads should seriously consider mixed-precision algorithms. The substantial 1.6× speedup can be obtained by carefully carrying out many of the operations in GMRES in single precision. The fact that we obtain this speedup on well-optimized code on a wide range of scales and on more than one architecture indicates that this is a realizable speedup for production scientific applications, not an artefact of some inefficiency or a particular run configuration. Further, if one uses half precision strategically for parts of operations in the blue region in algorithm 3, one can expect an even higher speedup. This will be addressed in future work. We do accept, however, that similar to other large-scale numerical benchmarks, the matrix is artificial and the actual speedup in applications will depend on the condition numbers and pseudo-spectra of the matrices. We introduced an option to run full-scale validation in our code and showed that the original benchmark's validation method sufficiently captures any loss of convergence rate.

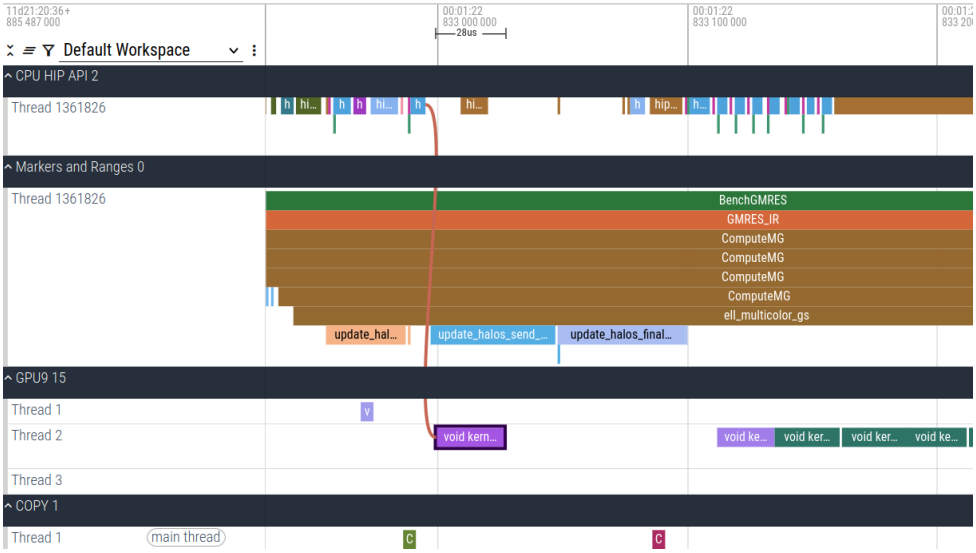
Critics may argue that HPG-MxP is redundant, since HPCG already exists and both are limited by memory bandwidth. However, we argue that HPG-MxP opens up a much bigger design space by introducing mixed-precision, nonsymmetric problems and GMRES (which has different memory utilization characteristics). Being a standardized benchmark, it allows a greater variety of scientists and engineers to engage with the issue it seeks to address, and can spur innovation in achieving greater performance for PDE-based simulation workloads.

In addition, we note that the mixed-precision GMRES-IR solver requires a lower-precision copy of the system matrix. This means its overall memory utilization is more than double-precision GMRES. In order to compensate for this, we should utilize a larger mesh size while running double-precision GMRES and it can perhaps achieve a somewhat higher throughput. The benchmark could be modified to take this into account. In some applications, however, this may not be relevant since the matrix-free variant of GMRES [12] or nonlinear GMRES [30] may be used. Only the low-precision matrix needs to be stored, instead of some approximate double-precision matrix, for preconditioning.

In closing, we also point out how helpful AMD's rocHPCG implementation [29] has been in achieving this demonstration of HPG-MxP performance. The fact that AMD open-sourced their implementation has accelerated further progress. In the same spirit, our code is also available open source and we provide details on building and running it in the reproducibility appendix.



(a) Fine-grid smoothing. The boxed operation is the GS kernel on the GPU.



(b) Coarsest grid smoothing. The boxed operation is the GS kernel on the GPU.

Figure 9: Traces of our GMRES-IR implementation in an 8-node benchmark run on Frontier. In each trace, there are 4 sections from top to bottom: 'CPU HIP API', 'Markers and Ranges', 'GPU' and 'COPY'. Purple bars in the 'GPU' section represent the interior Gauss-Seidel kernel. The consecutive orange, blue and gray-blue bars in the 'Markers and Ranges' section represent halo buffer and communications operations. In the 'COPY' section, the green bar represents the device-to-host copy of the send buffer, while the red bar after it is the host-to-device copy of the received data.

Acknowledgments

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.

References

[1] 2024. *NVIDIA Blackwell architecture technical brief.* Technical Report. NVIDIA. version 1.1.

[2] A. Abdelfattah, H. Anzt, A. Ayala, E. Boman, E. Carson, S. Cayrols, T. Cojean, J. Dongarra, R. Falgout, M. Gates, T. Gruetzmacher, N. Higham, S. Kruger, X. Li, N. Lindquist, Y. Liu, J. Loe, P. Luszczek, P. Nayak, D. Osei-Kuffuor, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. Tsai, I. Yamazaki, and U. M. Yang. 2021. *Advances in Mixed Precision Algorithms: 2021 Edition.* Technical Report LLNL-TR-825909. Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States). doi:10.2172/1814677

[3] Hartwig Anzt. 2024. *2.3.3.01- xSDK-Multiprecision Final Report for Subcontract Partner KIT.* Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States). doi:10.2172/2318788

[4] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Tsai, and Enrique S. Quintana-Ortí. 2022.

Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Trans. Math. Software* 48, 1 (March 2022), 1–33. doi:10.1145/3480935

[5] Hartwig Anzt, Björn Rocker, and Vincent Heuveline. 2010. Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms. *Computer Science - Research and Development* 25, 3 (2010). doi:10.1007/s00450-010-0124-2

[6] Allison H. Baker, Robert D. Falgout, Todd Gamblin, Tzanio V. Kolev, Martin Schulz, and Ulrike Meier Yang. 2012. Scaling Algebraic Multigrid Solvers: On the Road to Exascale. In *Competence in High Performance Computing 2010*, Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–226. doi:10.1007/978-3-642-24025-6_18

[7] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Hansol Suh, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2025. *PETSc/TAO Users Manual*. Technical Report ANL-21/39 - Revision 3.23. Argonne National Laboratory. doi:10.2172/2476320

[8] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. doi:10.1145/1654059.1654078

[9] Michele Benzi, Wayne Joubert, and Gabriel Mateescu. 1999. Numerical experiments with parallel orderings for ILU preconditioners. *Electron. Trans. Numer. Anal.* 8 (1999), 88–114. https://etna.math.kent.edu/volumes/1993-2000/vol8/abstract.php?vol=8%20&pages=88-114

[10] Achi Brandt. 1977. Multi-level adaptive solutions to boundary value problems. *Math. Comp.* 31, 138 (1977), 333–390. https://www.jstor.org/stable/2006422

[11] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. 2008. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Software* 34, 4 (2008). doi:10.1145/1377596.1377597

[12] Todd T. Chisholm and David W. Zingg. 2009. A Jacobian-free Newton–Krylov algorithm for compressible turbulent fluid flows. *J. Comput. Phys.* 228, 9 (2009), 3490–3507. doi:10.1016/j.jcp.2009.02.004

[13] Ewa Deelman, Jack Dongarra, Bruce Hendrickson, Amanda Randles, Daniel Reed, Edward Seidel, and Katherine Yelick. 2025. High-performance computing at a crossroads. *Science* 387, 6736 (2025), 829–831. doi:10.1126/science.adu0801

[14] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (Feb. 2016), 3–10. doi:10.1177/1094342015593158

[15] L. Giraud, J. Langou, and M. Rozloznik. 2005. The loss of orthogonality in the Gram-Schmidt orthogonalization process. *Computers & Mathematics with Applications* 50, 7 (2005), 1069–1075. doi:10.1016/j.camwa.2005.08.009 Numerical Methods and Computational Mechanics.

[16] William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. 2001. High-performance parallel implicit CFD. *Parallel Comput.* 27, 4 (2001), 337–362. doi:10.1016/S0167-8191(00)00075-2 Parallel computing in aerospace.

[17] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. 2018. The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques. In *Computational Science – ICCS 2018*, Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot (Eds.). Springer International Publishing, 586–600. doi:10.1007/978-3-319-93698-7_45

[18] Mark T. Jones and Paul E. Plassmann. 1993. A Parallel Graph Coloring Heuristic. *SIAM Journal on Scientific Computing* 14, 3 (1993), 654–669. doi:10.1137/0914041 arXiv:https://doi.org/10.1137/0914041

[19] Aditya Kashi. 2020. *Asynchronous fine-grain parallel iterative solvers for computational fluid dynamics*. phdthesis. https://escholarship.mcgill.ca/downloads/2f75rd57s

[20] Aditya Kashi, Hao Lu, Wesley Brewer, David Rogers, Michael Matheson, Mallikarjun Shankar, and Feiyi Wang. 2025. Mixed-precision numerics in scientific applications: survey and perspectives. arXiv:2412.19322 [cs.CE]

[21] Jennifer A. Loe, Christian A. Glusa, Ichitaro Yamazaki, Erik G. Boman, and Sivasankaran Rajamanickam. 2021. Experimental Evaluation of Multiprecision Strategies for GMRES on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 469–478. doi:10.1109/IPDPSW52791.2021.00078

[22] Michael Luby. 1986. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.* 15, 4 (1986), 1036–1053. doi:10.1137/0215074

[23] Maxim Naumov. 2011. *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*. Technical Report NVR-2011-001. NVIDIA.

[24] Maxim Naumov, Patrice Castonguay, and J. Cohen. 2015. *Parallel graph coloring with applications to the incomplete LU factorization on the GPU*. Technical Report NVR-2015-001. NVIDIA.

[25] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898718003

[26] Youcef Saad and Martin H. Schulz. 1986. GMRES - A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.* 7, 3 (1986).

[27] B.F. Smith, P.E. Bjørstad, and W.D. Gropp. 1996. *Domain decomposition - parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press.

[28] Brad Suchoski, Caleb Severn, Manu Shantharam, and Padma Raghavan. 2012. Adapting Sparse Triangular Solution to GPUs. In *2012 41st International Conference on Parallel Processing Workshops*. 140–148. doi:10.1109/ICPPW.2012.23

[29] Nico Trost. 2023. rocHPCG. https://github.com/ROCm/rocHPCG Advanced Micro Devices, inc..

[30] T. Washio and C.W. Oosterlee. 1997. Krylov subspace acceleration for nonlinear mulrigrid schemes. *Electronic Transactions on Numerical Analysis* 6 (Dec. 1997), 271–290.

[31] Ichitaro Yamazaki, Christian Glusa, Jennifer Loe, Piotr Luszczek, Sivasankaran Rajamanickam, and Jack Dongarra. 2022. High-Performance GMRES Multi-Precision Benchmark: Design, Performance, and Challenges. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 112–122. doi:10.1109/PMBS56514.2022.00015

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

This document describes the artifacts used to obtain the results of the article "Scaling the memory wall using mixed-precision - HPG-MxP on an exascale machine". The article describes a new state-of-the-art implementation of High Performance GMRES Mixed Precision (HPG-MxP) benchmark that achieves much higher performance than the reference implementation, performance analysis, speedup from the use of mixed precision, and performance on the full Frontier supercomputer, the first exascale machine.

Please note that there have been some bugfixes and minor improvements made to the source code since submitting the original manuscript, and these change the performance numbers slightly. At some scales, the performance numbers obtained by reproduction attempts may be slightly better than documented in the manuscript. However, the magnitudes of the changes are small and do not meaningfully affect the conclusions.

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

We provide a list of the main contributions of the paper below.

- **C1** We describe a state-of-the-art implementation of the HPG-MxP benchmark that achieves much higher performance than the reference implementation on large-scale GPU-based HPC systems.
- **C2** We show that with such an optimized implementation, a higher speedup than earlier reported is possible from a double-single mixed-precision solver on current-generation exascale systems.
- **C3** We report, for the first time, a full-system HPG-MxP run (9408 nodes) on the world's first exascale system, Frontier, at the Oak Ridge National Laboratory, USA, using our optimized codebase.
- **C4** We report some performance analysis of the benchmark code, particularly traces showing the achieved compute-communication overlap, the achieved memory bandwidth and performance relative to the roofline.
- **C5** We confirm that the standard validation that uses a small fixed problem size (on a single node) is sufficient to accurately penalize our mixed precision solver. This is achieved by introducing a full-scale validation that uses all available nodes and the full problem size.

A.2 Computational Artifacts

The artifact we provide is as follows.

- **A1** A permanent record of our HPG-MxP implementation along with plotting scripts is provided at https://doi.org/10.5281/zenodo.16943828, while code development is currently located at https://github.com/at-aaims/HPG-MxP.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 4
	C_2	Figure 5 Figure 6
	C_3	Figure 4 Sec. 4.1, line 751 Figure 7
	C_4	Figure 7 Figure 8 Figure 9
	C_5	Table 2

B Artifact Identification

We provide below details of how the artefact relates to the contributions, the expected reproduction time, setup of the computer system, execution of the code and analysis of the results. The instructions that follow enable the reader to set up the environment, build the code and run it on a high performance computing (HPC) system. Further, the steps to postprocess the data generated by the runs are outlined.

B.1 Computational Artifact A_1

Relation To Contributions

The artefact is the software repository developed for demonstrating optimized performance on the Frontier exascale system. It is the primary vehicle by which we obtain our findings and directly leads to all the results obtained, including absolute performance in GFLOP/s at different scales, speedups of the mixed-precision GMRES-IR over double-precision GMRES at different scales, time break-downs showing the proportion of time spent by the most important kernels, and traces showing the overlap of computation and communication.

Expected Results

The code should run and generate a `HPGMP-Benchmark_0.1_<date>_<time>.txt` file. This file contains the relevant performance data. On Frontier, the performance should be roughly 2000 GFLOP/s per node at large scales, dropping to 1838 GFLOP/s per node at full system scale.

Expected Reproduction Time (in Minutes)

The time required for an individual valid run, when the runtime specified is 1800s, is about 42 minutes on Frontier, at least up to 256 nodes or so. Half-time runs with runtime 900s, for node counts starting at 1024 nodes in our results, take about 32 minutes to complete. The difference is due to the problem set up tasks that are common to all variants of the code involved and are not officially timed.

	Frontier	Andes (GPU)
Max nodes	9408	9
GPU	AMD MI250x	NVIDIA K80
GPUs per node	8	2
GPU memory bandwidth	1600 GB/s	480 GB/s
GPU memory capacity	64 GB	24 GB
Network	HPE Cray Slingshot 11 200 Gb/s	Mellanox S8500 Series HDR Infiniband 200 Gb/s
CPU	AMD 7763, 64 cores	2x Intel Xeon E5-2695, 28 cores

Table 3: Hardware characteristics of target systems (note that on Frontier, 'GPU' means a single Graphics Compute Die (GCD))

	Frontier	Andes
C++ compiler	GCC 14.2.0	GCC 9.3.0
GPU toolchain	ROCm 6.2.4	CUDA 11.2.2
MPI	Cray MPICH 8.1.31, libfabric 1.22.0	OpenMPI 4.0.4

Table 4: Software used for building and running the code

Artifact Setup (incl. Inputs)

Hardware. The results reported in the article were mostly obtained on the Frontier exascale system, though those corresponding to figure 6 were obtained on Andes. The characteristics of these systems are documented in table 3.

Software. In general, building HPG-MxP requires

- a modern C++ compiler that supports the C++ 17 standard,
- a GPU toolkit, either CUDA or ROCm (CUDA 11.2, ROCm 6.2 and ROCm 6.4 have been tested so far),
- an MPI library, and
- CMake.

Table 4 shows the toolchains and libraries used for building the code on the respective systems. Note that ROCm and CUDA include drivers, compilers (`hipcc` or `nvcc`) and the ecosystem libraries rocPRIM, rocRAND rocBLAS, and rocSPARSE, or CUB, cuRAND, cuBLAS and cuSPARSE.

Datasets / Inputs. The matrix and vectors needed are generated within the code; there is no external data dependency. Apart from that, the code requires a few command line flags, denoting the local grid sizes and the requested runtime of the benchmarking phase: Eg., `–nx=320 –ny=320 –nz=320 –rt=1800` on Frontier.

Installation and Deployment. Installation consists of loading the required modules described above, running CMake to configure the build, and then building using the underlying build tool such as GNU Make. It is generally build out-of-tree and into a separate installation directory.

Profiling and tracing require some additional build flags to enable NVTX or rocTX annotations. The benchmark is also truncated to the most relevant parts for analysis.

Artifact Execution

After cloning or downloading the source code and building, execution primarily involves running the `xhpgmp` executable with the appropriate MPI and GPU-binding flags, with arguments to specify the local problem size, runtime and optionally the validation mode. To fully reproduce all results in the paper, the code needs to be run at different node counts. On Frontier, we ran the code at 1, 2 8, 64, 128, 512, 1024, 4096 and 9408 nodes. On Andes, we ran the code on 1, 2 and 8 nodes.

Artifact Analysis (incl. Outputs)

Running the `xhpgmp` executable will produce `HPGMP-Benchmark_0.1_<date>_<time>.txt` files. These contain all the performance data needed to generate Figures 4-7 in the paper (supporting contributions C_1-C_3) after post-processing. We provide the Python plotting scripts in the repository, in the `tools` directory. For each of the scripts, calling the script as `python3 script_name.py –help` outputs the purpose of the script and the arguments expected. We provide more detailed instructions on using these scripts in the artifact evaluation.

We used the ROCm Compute Profiler [2] tool for roofline analysis (Figure 8 in the paper, supporting contribution C_4). At the time of writing, a `rocprofiler-compute` module is available on Frontier. However, one can build the tool directly from source, which is the approach we initially took.

We used `rocprof` to generate traces and visualized these traces (`json` file) with Perfetto UI [3]. Figure 9 in the paper, supporting contribution C_4 is a snapshot of this trace vizualization.

The iteration counts and attained residual levels during validation can be read off from either the Slurm output file or the HPGMP-Benchmark output file. Table 2 (contribution C_5) in the paper is generated by collecting this data at different node counts using the full-scale validation.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

The following script can build HPG-MxP on Frontier, assuming one is currently in the root directory of the cloned repository.

```
module load PrgEnv-gnu/8.6.0 \
    libfabric/1.22.0 cray-mpich/8.1.31
module load rocm/6.2.4
module load googletest/1.14.0
module rm cray-libsci darshan-runtime
export LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH\
    :$LD_LIBRARY_PATH

mkdir build && cd build
CC=gcc CXX=g++ cmake -DHPGMP_ENABLE_HIP=ON \
```

[2] https://rocm.docs.amd.com/projects/rocprofiler-compute/en/latest/what-is-rocprof-compute.html
[3] https://ui.perfetto.dev/

```
    -DROCM_PATH=$ROCM_PATH \
    -DCMAKE_HIP_ARCHITECTURES=gfx90a \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=/path/to/install/hpgmp ..
make -j8
make install
```

Listing 1: Building on Frontier

```
module load gcc/9.3.0 cuda/11.2.2 openmpi/4.0.4
mkdir build && cd build
CC=gcc CXX=g++ cmake -DHPGMP_ENABLE_CUDA=ON \
    -DCMAKE_CUDA_ARCHITECTURES=37 \
    -DCMAKE_BUILD_TYPE=Release
    -DCMAKE_INSTALL_PREFIX=/path/to/install/hpgmp ..
make -j8
make install
```

Listing 2: Building on Andes

ROCm is detected mainly via the environment variable `ROCM_PATH`. Note that when using Cray compiler wrappers, it's best to point `CC` and `CXX` in CMake to the underlying compilers such as `g++` or `clang++`. CMake sometimes fails to find MPI otherwise.

To profile the application, we set the following additional flag at compilation time.

```
-DHPGMP_ENABLE_PROFILING=On
```

Listing 3: Extra compiling flag for profiling

This flag enables NVTx/ROCTx annotations and limits the number of GMRES calls to one (with `restart_length=30` multigrid cycles) for both the optimized and the reference implementations. This is sufficient for profiling purposes, while limiting the cost of profiling, given that there is a warm up run before the benchmark.

Artifact Execution

The listings below detail the steps and flags required to run the benchmark and obtain the results.

```
#SBATCH -t 0:45:00
#SBATCH -N 1024
#SBATCH --ntasks-per-node=8
#SBATCH --exclusive
#SBATCH --cpus-per-task=7
#SBATCH --gpus-per-task=1
#SBATCH -A acc123
#SBATCH -J m320_3-n1024
#SBATCH -o %x-%j.slout

module load PrgEnv-gnu/8.6.0 libfabric/1.22.0 \
    cray-mpich/8.1.31
module load rocm/6.2.4
module load googletest/1.14.0
module rm cray-libsci darshan-runtime
export LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH\
    :$LD_LIBRARY_PATH

HPCG_NX=320
HPCG_NY=320
HPCG_NZ=320
RUNTIME=1800
EXEC=/path/to/install/hpgmp/bin/xhpgmp
cd $SLURM_SUBMIT_DIR
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun -n $SLURM_NTASKS -c $SLURM_CPUS_PER_TASK \
    --gpus-per-task=1 --gpu-bind=closest \
    --exclusive \
    $EXEC --nx=$HPCG_NX --ny=$HPCG_NY --nz=$HPCG_NZ \
    --rt=$RUNTIME
```

Listing 4: Job submission script on Frontier

```
#SBATCH -t 0:55:00
#SBATCH -N 8
#SBATCH --ntasks-per-node=2
#SBATCH --exclusive
#SBATCH --cpus-per-task=14
#SBATCH --gpus-per-task=1
#SBATCH -p gpu
#SBATCH --mem=0
#SBATCH -A acc123
#SBATCH -J m256_160_128-n8
#SBATCH -o %x-%j.slout

HPCG_NX=256
HPCG_NY=160
HPCG_NZ=128
RUNTIME=1800
EXEC=/path/to/install/hpgmp/bin/xhpgmp

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun -n $SLURM_NTASKS -c $SLURM_CPUS_PER_TASK \
    --gpus-per-task=1 --gpu-bind=closest \
    -m block:cyclic --cpu-bind=sockets --exclusive \
    $EXEC --nx=$HPCG_NX --ny=$HPCG_NY --nz=$HPCG_NZ \
    --rt=$RUNTIME
```

Listing 5: Job submission script on Andes

The following commands can be used to generate PDF files of the roofline.

```
module load rocprofiler-compute/3.0.0
rocprof-compute profile --roof-only \
    --kernel-names --name xhpgmp \
    -- $EXEC --nx=$HPCG_NX --ny=$HPCG_NY --nz=$HPCG_NZ \
    --rt=$RUNTIME
```

Listing 6: ROCm Compute Profiler command for roofline analysis on Frontier

To collect the `rocprof` traces from multiple processes at once, we replace the original application executable with a helper script that includes a call to `rocprof` as follows.

```
srun -n $SLURM_NTASKS -c $SLURM_CPUS_PER_TASK \
    --gpus-per-task=1 --gpu-bind=closest \
    --exclusive ./helper_rocprof.sh
```

Listing 7: Scheduler command to collect traces on Frontier

```
rocprof --stats \
    --roctx-trace --sys-trace \
    -d ${SLURM_PROCID} \
    -o ${SLURM_PROCID}/results.csv \
    $EXEC --nx=$HPCG_NX --ny=$HPCG_NY --nz=$HPCG_NZ \
    --rt=$RUNTIME
```

Listing 8: rocprof command to collect traces on Frontier (content of helper_rocprof.sh)

This will generate traces for each process and write the data in the corresponding `$SLURM_PROCID` folder.

The default validation mode is the standard validation (single node) mode of Yamazaki et al. In order to obtain the results using full-scale validation to reproduce C_5 (table 2 in the revised manuscript), a slightly modified version of the job run script needs to be used; particularly, the flag `–validation_type=fullscale`.

```
#SBATCH -t 0:45:00
#SBATCH -N 1024
#SBATCH --ntasks-per-node=8
#SBATCH --exclusive
#SBATCH --cpus-per-task=7
#SBATCH --gpus-per-task=1
#SBATCH -A acc123
#SBATCH -J fullscale_valid-m320_3-n1024
#SBATCH -o %x-%j.slout

module load PrgEnv-gnu/8.6.0 libfabric/1.22.0 \
    cray-mpich/8.1.31
module load rocm/6.4.1
module load googletest/1.14.0
module rm cray-libsci darshan-runtime
export LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH\
    :$LD_LIBRARY_PATH

HPCG_NX=320
HPCG_NY=320
HPCG_NZ=320
RUNTIME=450
EXEC=/path/to/install/hpgmp/bin/xhpgmp
cd $SLURM_SUBMIT_DIR
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun -n $SLURM_NTASKS -c $SLURM_CPUS_PER_TASK \
    --gpus-per-task=1 --gpu-bind=closest \
    --exclusive \
    $EXEC --nx=$HPCG_NX --ny=$HPCG_NY --nz=$HPCG_NZ \
    --rt=$RUNTIME --validation_type=fullscale
```

Listing 9: Job submission script on Frontier for full scale validation

On a given system, it is currently a trial-and-error process to figure out the maximum possible local problem size. On Frontier, we use a local mesh size of 320^3, while on Andes we use $256 \times 160 \times 128$. For the multigrid method to work, each dimension must be divisible by 8.

Artifact Analysis (incl. Outputs)

Running the `xhpgmp` executable will produce `HPGMP-Benchmark_0.1_<date>_<time>.txt` files. We provide the following Python plotting scripts in the repository, in the `tools` directory.

- `parse_benchmark_output.py` With the HPGMP output file as the argument, this script parses the file and appends a row to a CSV file containing the time taken by different operations (SpMV, MG, etc.) and the GFLOP/s obtained by them.
- `plot_scaling.py` With the above generated CSV file as input, this script generates the weak scaling plot like the ones in Figure 4.
- `plot_speedups.py` With the CSV file as input again, this script generates speedup plots as in Figures 5 and 6 of the manuscript.
- `plot_stacked_breakdown.py` When called on the CSV file generated by the parser script, it generates the stacked bar charts. In response to reviewer comments, the pie charts in Figure 7 were replaced by a single stacked bar chart with different node counts along the x-axis.

The profiling results on Frontier (Figures 8 and 9 in the paper), supporting contribution C_4 can be obtained as follows:

- Running ROCm Compute Profiler will generate PDF files for the roofline. We built ROCm Compute Profiler from source and modified it to annotate the plot and make it more readable. We show the resulting roofline plot in Figure 8.
- Running `rocprof` with tracing will generate JSON files that can be visualized with Perfetto UI. We show a resulting trace in Figure 9. This is the trace from process 41 of 64 (8 Frontier nodes), which is at the center of the computational domain, and therefore, has the maximum number of neighbors.

We make an additional node about the `HPGMP-Benchmark_0.1_<date>_<time>.txt` files.

- The final reported mixed precision performance number is the field `GFLOP/s Summary::Total for benchmark`.
- The reported double precision performance number is the field `GFLOP/s Summary:: - Total (reference)`.
- For our implementation, the file also gives estimates of achieved memory bandwidth.
- Validation information is found in the 'Iterations Summary' section of the file, and also in the standard output (typically redirected to a Slurm output file).

Reproducibility Report

D Overview of Reproduction of Artifacts

The following table provides an overview of each computational artifact's reproducibility status. Artifact IDs correspond to those in the AD/AE Appendices.

Artifact ID	Available	Functional	Reproduced
A_1	•	•	partially
Badge awarded	yes	yes	no

E Reproduction of Computational Artifacts

E.1 Timeline

The artifact evaluation was conducted from Sept 05, 2025, to Sept 07, 2025."

E.2 Computational Environment and Resources

The experiments conducted for artifact evaluation were performed on

- NERSC Perlmutter
- A local server with amd=gfx90a and ROCm 6.2.0

E.3 Details on Artifact Reproduction

- The zip file was successfully downloaded from github.
- On both testbeds, the compilation failed initially
- This was fixed by changing constexpr to const in lines 34, 41, 57 of HPG-MxP-1.0.0/src/perf_counter.hpp
- On Perlmutter, the execution was successful with provided python scripts used to generate PNGs of scaling and speedups.
- The script plot_stacked_breakdown.py is not available in the artifacts while being mentioned in the AD/AE. This is due to the review process during which the pie charts were changed to stacked histograms but the changes did not reflect anywhere else.
- Although the exact plots were not reproduced, similar plots corresponding to Figuress 4-7 on Perlmutter were obtained.
- On the local gfx90a server, segmentation fault was encountered thus Figures 8-9 were not reproduced. Diligent effort was made by the reviewer to identify the source of segmentation fault, but that was unfortunately not possible in time.

Disclaimer: This Reproducibility Report was crafted by volunteers with the goal of enhancing reproducibility in our research domain. The time period allocated for the reproducibility analysis was constrained by paper notification deadlines and camera-ready submission dates. Furthermore, the compute hours in the shared infrastructure (e.g., Chameleon Cloud) available to the authors of this report were limited and restricted the scope and quantity of experiments in the review phase. Consequently, the inability to reproduce certain artifacts within this evaluation should not be interpreted as definitive evidence of their irreproducibility. Limitations in the time allocated to this review and the compute resources available to the reviewers may have prevented a positive outcome. Furthermore, reviewers assess the reproducibility of the artifacts provided by the authors; however, they are not accountable for verifying that the artifacts support the main claims of the paper.