# What Will the Grace Hopper-Powered Jupiter Supercomputer Bring for Sparse Linear Algebra?

Yu-Hsiang Tsai
Technical University of Munich
Heilbronn, Germany
yu-hsiang.tsai@tum.de

Mathis Bode
Juelich Supercomputing Centre
Juelich, Germany
m.bode@fz-juelich.de

Hartwig Anzt
Technical University of Munich
Heilbronn, Germany
Innovative Computing Laboratory
Knoxville, United State
hartwig.anzt@tum.de

## Abstract

The first exascale supercomputer in Europe, JUPITER, is currently being built using the NVIDIA Grace Hopper superchips as main building blocks. JUPITER is designed to provide computing power for both data-driven (AI) workloads and numerics-based simulation workloads. For both workload types, and particularly for PDE-based simulations, high-performance sparse linear algebra operations are crucial. In this paper, we analyze the performance levels that sparse linear algebra operations can achieve on the JUPITER supercomputer and identify algorithmic modifications that can improve performance by acknowledging the Grace Hopper architecture.

## CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**; **Parallel computing methodologies**.

## Keywords

GPU, sparse linear algebra, Grace Hopper, supercomputer

**ACM Reference Format:**
Yu-Hsiang Tsai, Mathis Bode, and Hartwig Anzt. 2026. What Will the Grace Hopper-Powered Jupiter Supercomputer Bring for Sparse Linear Algebra?. In *SCA/HPCAsia 2026: Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region (SCA/HPCAsia 2026), January 26–29, 2026, Osaka, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3773656.3773691

## 1 Introduction

Deep learning and computational fluid dynamics (CFD) simulations require large and high-performance supercomputers. As part of the US Exascale Computing Project [15], the US has already commissioned three supercomputers exceeding the Exascale threshold in the High Performance Linpack (HPL [9]) benchmark. Europe is about to exceed the Exascale threshold with the JUPITER supercomputer that is currently being deployed at the Juelich Supercomputing Centre in Germany, and will feature roughly 6,000 nodes each containing 4 NVIDIA Grace Hopper superchips (GH200)[5]. Already with only a portion of the supercomputer assembled, JUPITER secured the 4th place in the November 2025 TOP500 list[8]. While AI workloads have been at the center of the NVIDIA Grace Hopper superchip design, the GH200-based JUPITER supercomputer will also be used for traditional simulation workflows based on the discretization of (partial) differential equations. Particularly for those simulation workflows, scalable sparse linear algebra functionality is a key component, often determining the performance of the scientific applications.

It is well-known that supercomputers are toothless tigers without algorithms and software stacks that are capable of translating the computing power of the hardware into application performance. For this reason, it is important that even before commissioning supercomputers for full access, central building blocks that are the backbone of many applications are optimized, and the system-specific optimization tricks are documented for other users.

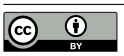
In this paper, we investigate the performance of key sparse linear algebra functionalities of the Ginkgo [6] open-source library for the JUPITER supercomputer and suggest optimization steps to boost the performance for many-GPU execution.

The rest of the paper is structured as follows: We initially introduce in Section 2 the JUPITER supercomputer, motivate the choice of the Ginkgo software stack, and list the hardware and software environment of the JUPITER supercomputer in Section 3. In Section 4 and its subsections, we discuss the performance levels we achieve for Ginkgo's single-GPU sparse matrix vector product (SpMV) kernel and suggest optimizations to match the performance of NVIDIA's implementation in the cuSPARSE library. Building upon the single-node SpMV kernel, we propose a method to hide the communication in the distributed SpMV algorithm. We interleave the algorithm discussions with weak and strong scaling experiments to allow the reader to quickly connect the algorithmic changes with the performance implications. Applying these optimizations to Ginkgo's functionality allows us to also analyze its impact on real-world applications in strong scaling of a full-fledged Conjugate Gradient (CG) and generalized minimal residual (GMRES). We summarize our observations in Section 5.

## 2 Background

### 2.1 JUPITER

JUPITER, the "Joint Undertaking Pioneer for Innovative and Transformative Exascale Research", is deployed in the Juelich Supercomputing Centre in Germany. It will aggregate about 6,000 nodes, each hosting four Grace Hopper superchips (GH200) as visualized in Figure 1. Each GPU has 96 GB 4TB/s HBM3 memory. The GPUs can communicate at a rate of 150 GB/s per direction. Each CPU contains




*SCA/HPCAsia 2026, Osaka, Japan*

ACM ISBN 979-8-4007-2067-3/2026/01
https://doi.org/10.1145/3773656.3773691

72 cores and has 120 GB 500GB/s LPDDR5X memory. The Grace CPU can communicate with the Hopper GPU at a bandwidth of 450 GB/s per direction. The CPUs can communicate among themselves at a rate of 100 GB/s per direction. The distinct nodes of JUPITER are connected with InfiniBand NDR (200 Gb/s or 25 GB/s per direction per chip). The network topology is based on 25 DragonFly+ groups.

## 2.2 Ginkgo

Ginkgo is a GPU-centric math library [6]. While other high-performance numerical linear algebra libraries exist, with PETSc [3], Trilinos [17], and SuperLU [13, 14] being among the most popular examples, we selected Ginkgo due to its GPU-centric modern C++ design, and the fact that Ginkgo is being used by application codes running on JUPITER. In some detail, Ginkgo supports a wide range of hardware architectures using a backend model that allows for kernel performance engineering in the vendor-native language. This is an advantage when optimizing for a specific GPU architecture and leveraging the latest GPU features [6, 7, 18].

# 3 Hardware and Software Environment

At the time of writing, the JUPITER supercomputer is in its pre-production phase, during which hardware and software have not yet converged to the final setting. Against this background, the experimentation and performance results presented in this paper contribute to both software and performance optimization on the one hand and system engineering on the other, aiming to converge on a performance-stable hardware-software ecosystem. At the same time, it is important to consider the hardware and software configuration when reproducing the results at a later point in time. The software stack we use in the experimentation is based on the `Stage/2025` modules available on the JUPITER supercomputer. We use GCC 13.3.0, CUDA 12.6.2, and OpenMPI 5.0.5. At the time of writing, the UCX protocol configuration is not yet completed, and we are excluding `gdr_copy` from all experiments.

For the performance evaluation, we use scalable test cases arising from a finite difference discretization of a Laplace operator. Particularly, we use the following test problems:

(1) five-point (5pt) stencil: 2D discretization of the Laplace operator, connection with itself and its four direct neighbors.
(2) seven-point (7pt) stencil: 3D discretization of the Laplace operator, connection with itself and its six direct neighbors.
(3) nine-point (9pt) stencil: 2D discretization of the Laplace operator, connection with itself and its eight neighbors.
(4) twenty-seven-point (27pt) stencil: 3D discretization of the Laplace operator, connection with itself and its twenty-six neighbors.

For any of the k-point stencils, all off-diagonal entries are set to `-1.0` and the diagonal is set to `k - 1.0`. The stencil geometry is a square for the 2D cases and a cube for the 3D cases, respectively.

Given that stencil discretizations result in a balanced nonzero distribution across the rows, we focus on using Ginkgo's classical Compressed Sparse Row (CSR) format for the local contributions and Ginkgo's Coordinate format (COO) for the non-local contributions. All numerical computations use IEEE754 double precision (64-bit arithmetic).

JUPITER's GH200 architecture superchips feature tightly connected CPU DDR and GPU HBM3 memories serving the Grace ARM cores and the Hopper CUDA cores, respectively. The tight coupling, however, allows the CUDA cores to also access data in the DDR (and vice versa). While a comprehensive evaluation of the performance characteristics of different memory accesses is available in [10], we focus here on the cost of streaming data for memory-bound kernels. This scenario characterizes sparse linear algebra functionality with coalesced memory access, e.g., solving a large structured FEM/FD problem like the Finite Difference stencil discretizations we use as test problems. Figure 2, following the roofline concept[19], presents the experimentally-evaluated performance roofs of the Hopper GPU in the GH200 superchip accessing data either in HBM3 memory or the DDR memory (labeled fp64/fp32 gh200_host).

The roofline plot relating the performance of a kernel to its arithmetic intensity (ratio between floating point operations and memory accesses) reveals that kernels with an arithmetic intensity smaller than 56 operations-per-value are memory-bound when accessing data in HBM3 memory. When accessing data in DDR memory, operations are memory-bound until the arithmetic intensity exceeds 1,000 operations per value. We note that expressing the arithmetic intensity in the operations-per-value metric (not relating to bytes) makes these turnaround points independent of the floating point format used. Evaluating the rooflines against the sparse linear algebra characteristics, we conclude that 1.) all sparse linear algebra functionality we address (SpMV, iterative solvers) are memory bound on the GH200 architecture; and 2.) accessing data in DDR vs HBM3 is an order of magnitude slower, and hence, all data should be kept in HBM3 memory.

GH200's feature of accessing data in DDR from the Hopper GPU eases programmability, but the performance penalty can be significant. We note that NVIDIA implements a memory management model that, in most scenarios, succeeds in automatically managing the data locality without performance losses. However, there is no fine control, so we cannot always force the memory location and memory access path when we measure the average performance. After a few accesses, it might decide to copy the data from CPU to GPU implicitly such that the GPU kernel does not need to grab the data through CPU. This behavior makes sense in practice. However, analyzing performance becomes more complex because different runs may use different memory locations or access data via different paths. We refer [10] for more detailed analysis. In this paper, we store all testing data in GPU memory to ensure that no additional effects are introduced by this memory behavior.

# 4 Performance Assessment and Optimization

We perform the following experiments only on GPU memory. The GPU memory is large enough to contain all problem data.

## 4.1 Single-GPU SpMV Performance

In the past, Ginkgo's SpMV kernels have been optimized for NVIDIA's V100 and A100 architectures; both the CSR and COO SpMV kernels have been competitive with the SpMV kernels shipped with NVIDIA's cuSPARSE library. However, running the same kernels on

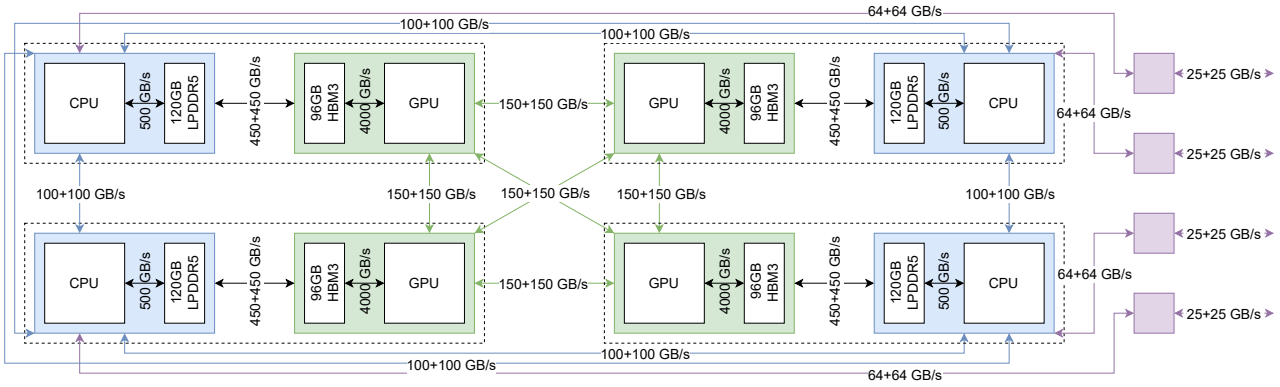



**Figure 1: The design of a Grace Hopper node in JUPITER[1].**

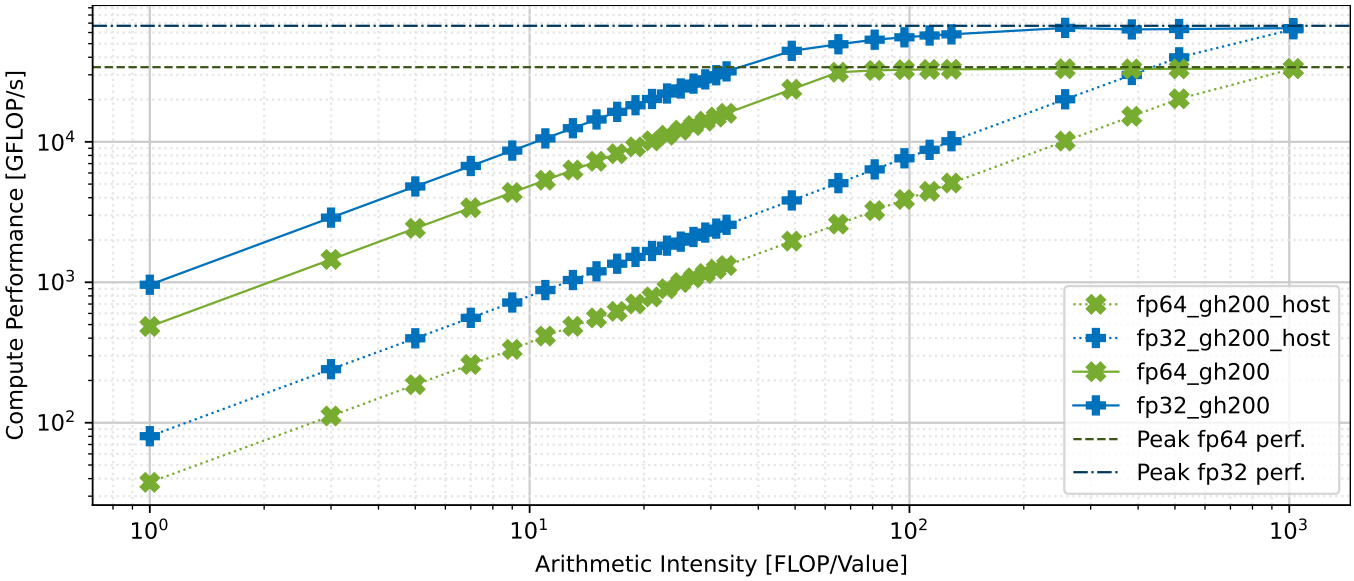



**Figure 2: Roofline of the Hopper GPU in an NVIDIA GH200 superchip accessing data either in HBM3 or DDR memory.**

the GH200 GPU, they perform inferior to their counterparts available in NVIDIA's cuSPARSE library. For a detailed performance analysis, we pick the 27pt stencil test case of size $10^7$ and investigate the execution with the Nsight Compute profiler. The analysis reveals that Ginkgo's generalized kernel is harming the performance of the kernels on the GH200 chip. Ginkgo's SpMV kernels are generalized in two aspects: 1) Ginkgo's SpMV kernel is mixed-precision by design, which means the input objects can be in different precision formats, and the kernel automatically converts to the highest precision and performs the computations. 2) Ginkgo's SpMV kernels are generalized in terms of the vector, i.e., a multi-vector (tall-and-skinny dense matrix) is also accepted by the SpMV kernel (becoming SpMM, then). The Nsight profiler analysis reveals that the arithmetic operations needed for 1) and 2) are the reason for Ginkgo SpMV kernels being inferior to the cuSPARSE SpMV kernels. The operations necessary for 1) and 2) are arithmetic operations ensuring the read and write bounds in Ginkgo's multi-vector SpMV kernel and selecting the matching floating point, respectively, and did not degrade performance on NVIDIA's previous architecture generations. This is the first time we have observed that the memory-bound SpMV kernel is impacted by index computations, even though the arithmetic units are not saturated. A possible explanation is that the cycles from these operations are very long, such that the kernel is not always reading data from the GPU global memory.

To improve the performance of Ginkgo's SpMV kernels for the GH200 architecture, we need to create a new kernel overload that only accepts a single vector as input and adjusts the thread oversubscription factor in the kernel launch. Thanks to Ginkgo's flexible portability design, specializing the kernel just in one specific kernel does not hurt the portability of Ginkgo. We easily improve the performance on our side without hurting the other backend performance.

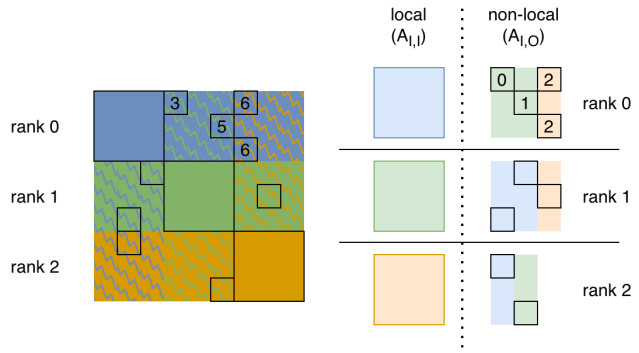



**Figure 3: Distributed matrix data layout. We uniformly distribute the $9 \times 9$ matrix to three ranks. On the left side, the solid-colored background indicates the storage location - blue, green, and orange colors corresponding to ranks 0, 1, and 2, respectively. The zigzag line indicates the column distribution for the different ranks. When the color of the zigzag line differs from the solid background color, it represents the non-local matrix of the rank. The right part provides more details on how Ginkgo stores the matrix: Ginkgo stores the local and non-local matrix separately. Ginkgo re-indexes the columns such that only the necessary right-hand side values are retrieved. The number in the block indicates the column index. 3, 5, and 6 of rank 0's part become 0, 1, and 2 when we assemble the non-local matrix of rank 0.**

Applying these changes to Ginkgo's CSR and COO SpMV kernels, accelerates the kernels by 1.13× and 1.3×, respectively. In comparison to NVIDIA's cuSPARSE library, our classical CSR SpMV kernel is now around 4% faster than cuSPARSE, our COO SpMV kernel is about 1.5% slower than cuSPARSE. We acknowledge that this performance engineering effort focused on the finite difference stencil matrices. However, as the optimization steps only reduce the number of arithmetic operations, we do not expect performance drawbacks for other matrices.

## 4.2 Overlapping Communication and Computation

Ginkgo's distributed matrix is a 1-D row distribution. We use the same distribution for columns which separates the matrix into a local part (local matrix) and a non-local part (non-local matrix). We visualize this concept in Figure 3 for the toy case of a $9 \times 9$ matrix and three ranks.

The distributed vectors follow the same distribution. Using this distribution, the distributed SpMV kernel becomes

$$\begin{aligned} A_{I,:} \times x &= (A_{I,I}(local) + A_{I,O}(non\text{-}local))x \\ &= A_{I,I} \times x_I + A_{I,O} \times x_O \end{aligned}$$

where $I$ is the set of indices that belong to one rank and $O$ is the set of indices not in $I$. $A_{I,I} \times x_I$ does not require any communication as all matrix and vector values are local, but $x_O$ requires communication before performing the SpMV kernel. In Figure 3, our non-local matrix is re-indexed such that not all values of $x_O$ are communicated. In some detail, only the subset of $x_O$ necessary for the non-local matrix is communicated. We visualize this concept in Figure 4. Before initializing the communication, the data from each rank needs to be prepared in the sense of gathering the entries required from $x_O$ into a single array. After preparation, the necessary data is communicated using `i_all_to_all_v` to exchange the data. During the communication of the $x_O$ values, the local computations $A_{I,I} \times x_I$ can be executed to overlap and hide the communication. For easier reference, we will call the operations $A_{I,I} \times x_I$ the *local SpMV* and the operations $A_{I,O} \times x_O$ the *non-local SpMV*.

Unfortunately, OpenMPI[11] supports GPU-aware MPI but does not support stream-aware MPI, so we need to synchronize before calling the asynchronous MPI routine (`i_all_to_all_v`). We need to use `i_all_to_all_v` because the size of each rank is not equal like Figure 4.

The original Ginkgo's workflow was:

(1) prepare data for $x_O$
(2) synchronize the stream
(3) call the asynchronous MPI routine
(4) submit the kernel for local SpMV $A_{I,I} \times x_I$
(5) wait for the MPI routine
(6) submit the kernel for non-local SpMV $A_{I,O} \times x_O$

However, using NVIDIA's Nsight profiler, we identified a gap in GPU activity. In the original workflow listed above, the local SpMV kernel $A_{I,I} \times x_I$ is submitted after synchronization, so the GPU inactivity stems from the kernel submission, synchronization, and MPI message submission overhead. We visualize the original workflow along with time measurements in workflow in Figure 5. This experiment again uses the 27pt stencil test case of size $10^7$ and a setup of 8 GH200 GPUs to include both node-local and inter-node communication. Note that the length of the bars does not scale to the timings; the bars are adjusted to visualize the workflow. In this setting, the gap between GPU activities averages 54 $\mu s$, which is about 24% of the 224 $\mu s$ SpMV execution time. Because communication can overlap with computation, we can shorten the overall execution time by narrowing the gap.

To reduce the GPU inactivity, we change how we synchronize in the kernel execution. In the new workflow, after launching the kernel that prepares the data for $x_O$, we record an event into the CUDA stream and immediately submit the local SpMV kernel, even before calling the MPI communication, but call the MPI communication only after event synchronization. The SpMV workflow becomes:

(1) prepare data for $x_O$
(2) record event into the stream
(3) submit the kernel for local SpMV $A_{I,I} \times x_I$
(4) wait for the event
(5) call the asynchronous MPI routine
(6) wait for the MPI routine
(7) submit the kernel for non-local SpMV $A_{I,O} \times x_O$

By doing so, the local SpMV kernel computing $A_{I,I} \times x_I$ is not affected by synchronization from preparing data. While changing the order will make MPI call face additional overhead from event recording and SpMV kernel submission, the SpMV $A_{I,I} \times x_I$ is likely able to fully hide the communication. If the communication can not be hidden by the local SpMV, the communication is dominating the kernel execution. Figure 6 visualizes the new workflow. The measurements confirm that the new strategy narrows the GPU

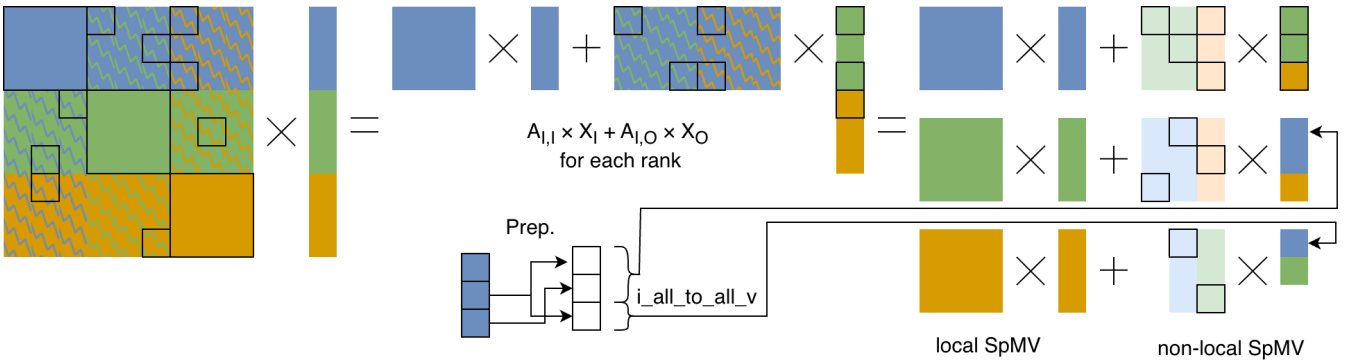



**Figure 4: Distributed SpMV. The left-hand side visualizes the original SpMV. The middle and upper parts sketch the work of each rank. The right-hand side visualizes the local SpMV and non-local SpMV using the matrix structure introduced in Figure 3. The right-hand side of non-local SpMV requires the data preparation and the MPI `i_all_to_all_v` routine as sketched in the middle and lower parts.**

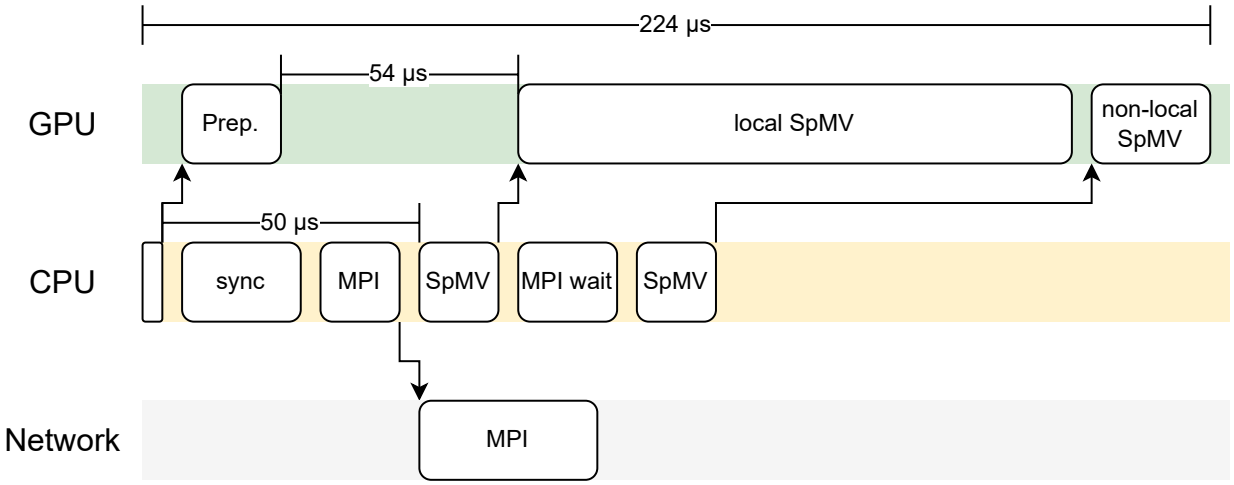



**Figure 5: Original workflow: synchronization before MPI and local SpMV. Note that the bar lengths do not scale to the execution time. The runtimes are measured with the Nsight system profiler on 8 Grace Hopper superchips.**

inactivity gap, and the overall SpMV execution is accelerated by 1.2×.

To accelerate the communication, we investigate the use of NCCL, NVIDIA's stream-aware MPI collective. We utilize NCCL as the communication library in a full-fledged Conjugate Gradient (CG) solver that relies on Ginkgo's distributed SpMV kernel. We choose a full-fledged CG in this analysis, as replacing OpenMPI `all_reduce` with NCCL in the Ginkgo library when compiling with NCCL support for the distributed SpMV. The results visualized in Figure 7 are not conclusive whether NCCL should be preferred over OpenMPI. As NCCL does not provide `i_all_to_all_v` functionality and limits the portability of the Ginkgo software stack, we decide to continue relying on OpenMPI for the time being.

### 4.3 SpMV Weak Scaling Analysis

We next aim to quantify the performance boost achieved by reducing the arithmetic operations in the local SpMV kernel (Section 4.1) and optimizing the communication in the distributed SpMV (Section 4.2). Figure 8 visualizes the results of a weak scaling experiment in which we compare against Ginkgo's original distributed SpMV kernel. The results indicate that the architecture-specific optimizations result in a 1.06× ∼ 1.44× speedup.

To assess the parallel efficiency, we collect the distributed SpMV performance on different stencil configurations (2D 5pt/9pt and 3D 7pt/27pt) with different local sizes from 100 to $10^7$ on up to 2048 nodes (8192 Grace Hopper superchips). Figure 9 visualizes the parallel efficiency with respect to one node (4 Grace Hopper superchips). For large local problem sizes, the parallel efficiency remains high due to the significant local workload. For very sparse

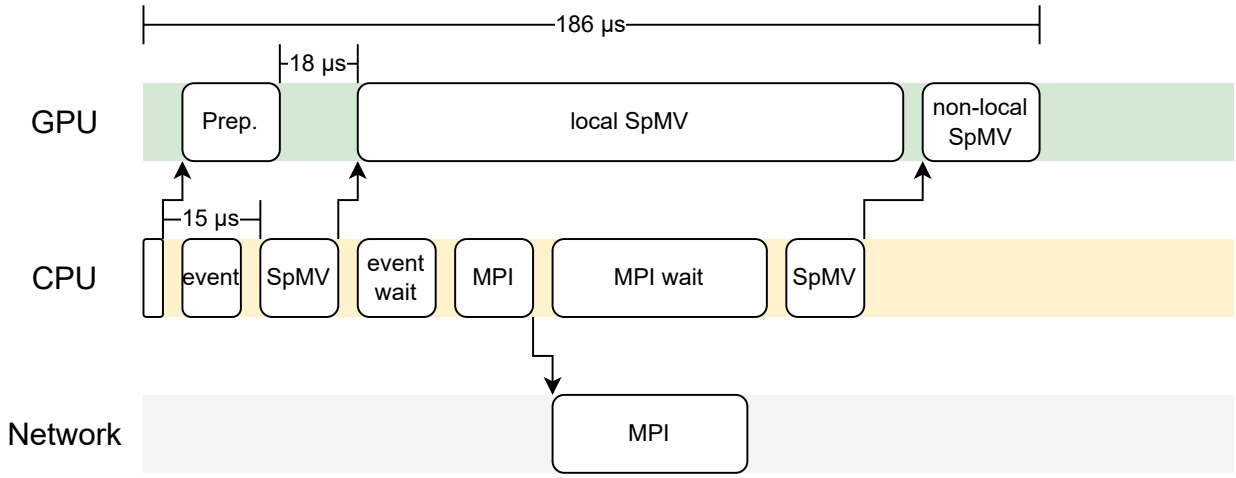



**Figure 6: New workflow: a CUDA event allows for submitting the local SpMV prior to the MPI call, thereby hiding the communication with the local SpMV. Note that the bar lengths do not scale to the execution time. The runtimes are measured with the Nsight system profiler on 8 Grace Hopper superchips.**

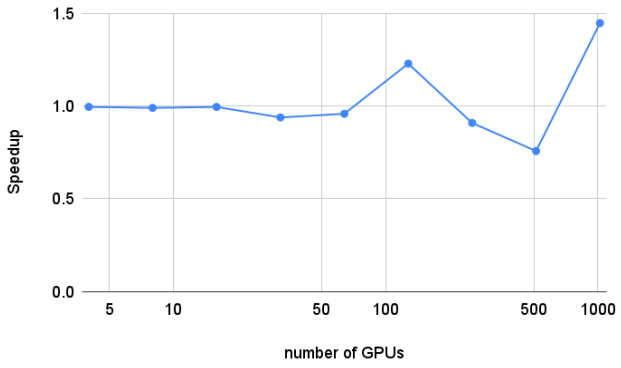



**Figure 7: OpenMPI `all_reduce` Speedup over NCCL on 1 up to 256 nodes (4 to 1024 Grace Hopper superchips). It is based on distributed Conjugate Gradient solving 3D Laplace problem discretized with a 27pt finite difference stencil (163M size, 27pt stencil).**

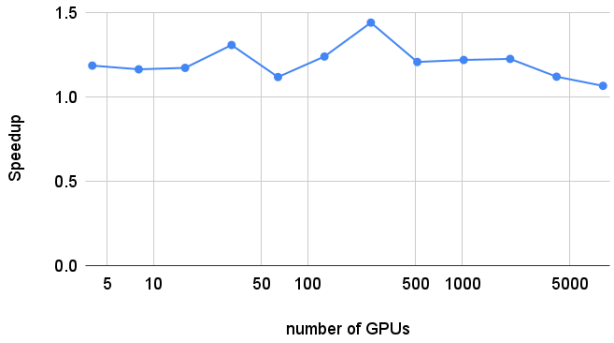



**Figure 8: Speedup of new distributed SpMV over the original code on up to 2048 nodes (8192 Grace Hopper superchips). It is based on weak scaling experiments on the 27pt stencil test case with a local problem size of $10^7$.**

problems (7pt stencil), the parallel efficiency is low because of a high communication-to-computation ratio. For the 27pt stencil test case with a local size of $10^7$, the parallel efficiency remains at around 80% on up to 2048 GPUs. Using the same local problem size, other discretizations (resulting in fewer computations) maintain 80% parallel efficiency for up to 256 or 512 GPUs. The efficiency then drops to 60% parallel efficiency for higher GPU counts.

Figure 10 visualizes the execution rates achieved in the weak scaling experiments using the metric $FLOP/s = 2 \times nnz/runtime$. For the local size $10^7$, Ginkgo's distributed SpMV achieves for the 27pt, 5pt, 7pt, and 9pt stencil discretizations up to 1.9, 1.2, 1.0, and 1.4 PFLops, respectively.

## 4.4 Performance Development over NVIDIA GPU Generations

In preparation for JUPITER, the Juelich Supercomputing Centre deployed the significantly smaller JURECA[2] system which, as a prototype, shares design characteristics with JUPITER. The system is modular in the sense that it features CPU-only nodes and GPU-accelerated nodes to reflect the application requirements. The GPU-accelerated compute nodes are equipped with 4 NVIDIA A100 GPU[4]. Each GPU has 40GB 1,555 GB/s HBM2 memory. The nodes of JURECA are connected with InfiniBand HDR (100 Gb/s or 12.5 GB/s per direction per chip).

In Figure 11, we compare the performance numbers achieved on JUPITER with those obtained on the JURECA supercomputer, which features 4 NVIDIA A100 GPUs. We use the same libraries

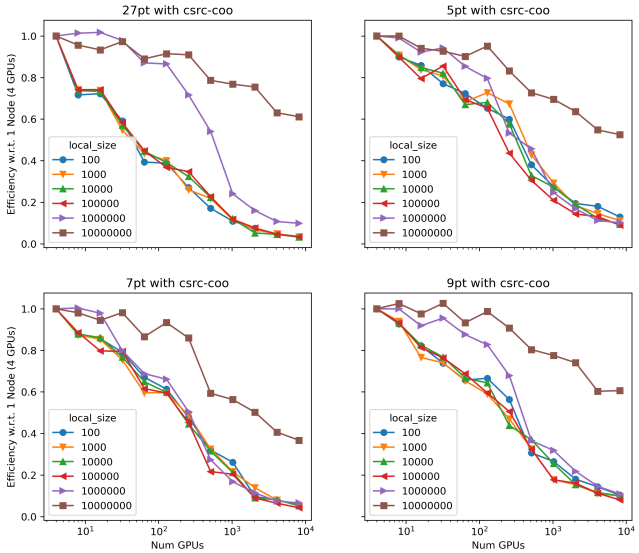



**Figure 9: Distributed SpMV weak scaling. Parallel efficiency analysis on up to 2048 nodes (8192 Grace Hopper superchips).**

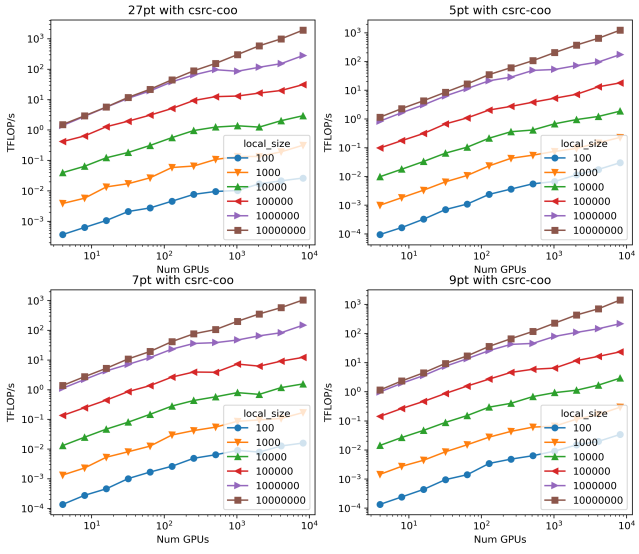



**Figure 10: Distributed SpMV weak scaling on up to 2048 nodes (8192 Grace Hopper superchips).**

and compilers in JURECA as JUPITER, as Section 3 mentioned. We limit the analysis to the 27pt stencil test case with a local size of $10^6$ to reflect the smaller memory capacity of the A100 GPU. Figure 11 reveals that the hardware improvement provides a 1.2× ∼ 1.45× speedup when moving from the A100 GPU system to the GH200 GPU system. When combining the hardware advancement with algorithmic changes as detailed in Section 4.1 and Section 4.2, the speedup grows to 2.4× ∼ 2.7×.

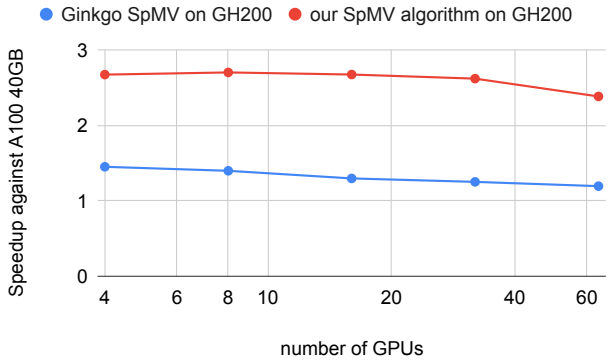



**Figure 11: Speedup coming from hardware advances (GH200 vs A100) and additional algorithmic optimization. Weak scaling test case: 27pt stencil with local size $10^6$.**

## 4.5 Strong Scaling - Solver

Subsequently to the SpMV optimization and performance assessment, we next investigate a full-fledged Conjugate Gradient[12] (CG) algorithm and a generalized minimal residual method[16] (GMRES) solver, both using the optimized distributed SpMV as a central building block. For the CG solver, the SpMV is the runtime-dominating component. The GMRES solver, other than the CG, which does not rely on short recurrences, has a communication-intensive orthogonalization step that orthogonalizes the new Krylov basis vector against all previous Krylov basis vectors, and a local least-square problem to find the solution approximation in the Krylov subspace. Depending on the restart parameter, the SpMV kernel or the orthogonalization step dominates the cost of the GMRES solver. For the strong scaling analysis, we maximize the problem size of the distinct tests to fit into the memory of the GPUs of the base case configuration. In some detail, we consider three 27pt stencil discretization test cases with size 163M unknowns (fits into the memory of 4 GH200 GPUs), 327M unknowns (fits into the memory of 8 GH200 GPUs), and 1655M unknowns (fits into the memory of 16 GH200 GPUs). In Figure 12, we visualize the strong scaling experiments. For the CG solver (left-hand-side), linear scalability is preserved for 3 resource-doubling steps, and runtime still decreases for a fourth resource-doubling step (total of 16× resource increase). After that, the runtime no longer decreases, but even increases due to the communication overhead. For the GMRES solver, scalability is worse due to the increased communication and the least-square problem solver. Overall, the GMRES solver runtime decreases only for up to 8× resource increase.

## 5 Conclusion

We have evaluated sparse linear algebra functionality and its scalability on the JUPITER supercomputer that is being deployed as Europe's first Exascale system at the Juelich Supercomputing Centre. We have demonstrated that executing kernels and algorithms initially optimized for NVIDIA's V100 and A100 architectures can result in sub-optimal performance on JUPITER's GH200 GPUs. We have furthermore detailed which algorithmic changes help to boost

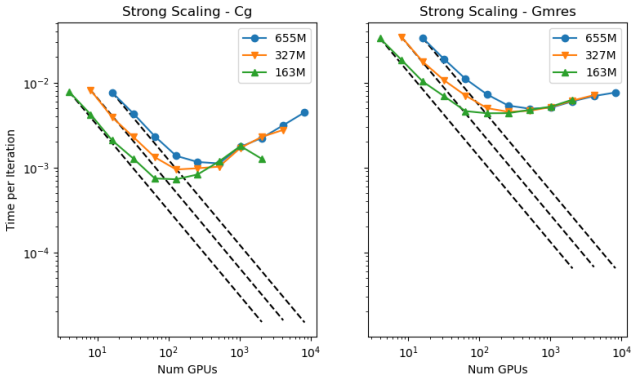



**Figure 12: CG and GMRES strong scaling result on 27pt stencil. The legend is matrix size. The base case of green/orange/blue runs on 4/8/16 GPUs, respectively.**

the performance on the GH200 GPUs. Aside from the single-GPU case, we also assessed the performance of the distributed execution of sparse linear algebra functionality on the GH200-powered supercomputer and propose design changes to Ginkgo's distributed SpMV workflow to boost performance. We demonstrated that when comparing with supercomputers based on the preceding A100 GPUs, the GH200-based JUPITER supercomputer in combination with the architecture-specific kernel optimizations can deliver up to 2.7× speedups for sparse linear algebra functionality.

## Acknowledgments

This research is supported by the Inno4Scale project under Inno4scale-202301-099. Inno4Scale has received funding from the European High Performance Computing Joint Undertaking (JU) under grant agreement No 101118139. The JU receives support from the European Union's Horizon Europe Programme. The authors acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputers at Jülich Supercomputing Centre (JSC). This project received access to the JUPITER supercomputer, which is funded by the EuroHPC Joint Undertaking, the German Federal Ministry of Research, Technology and Space, and the Ministry of Culture and Science of the German state of North Rhine-Westphalia, through the JUPITER Research and Early Access Program (JUREAP).

## References

[1] [n. d.]. JUPITER Technical Overview. https://www.fz-juelich.de/en/ias/jsc/jupiter/tech.
[2] [n. d.]. JURECA Configuration. https://apps.fz-juelich.de/jsc/hps/jureca/configuration.html.
[3] 2021. PETSc/Tao: Home Page. https://www.mcs.anl.gov/petsc/index.html.
[4] 2025. NVIDIA A100 specification. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf. Accessed: 19-09-2025.
[5] 2025. NVIDIA Grace Hopper Superchip datasheet. https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip. Accessed: 19-09-2025.
[6] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S Quintana-Ortí. 2022. Ginkgo: A modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–33.
[7] Terry Cojean, Yu-Hsiang Mike Tsai, and Hartwig Anzt. 2022. Ginkgo—A math library designed for platform portability. *Parallel Comput.* 111 (2022), 102902.
[8] Jack Dongarra and Piotr Luszczek. 2011. TOP500. In *Encyclopedia of parallel computing*. Springer, 2055–2057.
[9] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.
[10] Luigi Fusco, Mikhail Khalilov, Marcin Chrapek, Giridhar Chukkapalli, Thomas Schulthess, and Torsten Hoefler. 2024. Understanding data movement in tightly coupled heterogeneous systems: A case study with the Grace Hopper superchip. *arXiv preprint arXiv:2408.11556* (2024).
[11] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.
[12] Magnus R Hestenes, Eduard Stiefel, et al. 1952. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards* 49, 6 (1952), 409–436.
[13] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Software* 29, 2 (June 2003), 110–140. doi:10.1145/779359.779361
[14] Xiaoye S Li, Paul Lin, Yang Liu, and Piyush Sao. 2023. Newly released capabilities in the distributed-memory SuperLU sparse direct solver. *ACM Trans. Math. Software* 49, 1 (2023), 1–20.
[15] Paul Messina. 2017. The exascale computing project. *Computing in Science & Engineering* 19, 3 (2017), 63–67.
[16] Youcef Saad and Martin H Schultz. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing* 7, 3 (1986), 856–869.
[17] The Trilinos Project Team. [n. d.]. *The Trilinos Project Website.*
[18] Yu-Hsiang M Tsai, Terry Cojean, and Hartwig Anzt. 2023. Providing performance portable numerics for Intel GPUs. *Concurrency and Computation: Practice and Experience* 35, 20 (2023), e7400.
[19] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.