Institute of Theoretical Informatics (ITI)
Department of Computer Science
Karlsruhe Institute of Technology

Portable Mixed Precision Algebraic Multigrid on High Performance GPUs

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von

Yu-Hsiang Tsai
geboren in Kaohsiung City, Taiwan

Tag der mündlichen Prüfung: 2024 February 12

1. Referent: Prof. Dr. Hartwig Anzt
2. Referent: Prof. Dr. Wolfgang Karl
3. Referent: Prof. Dr. Enrique S. Quintana-Ortí

Abstract

Multigrid methods are algorithms for solving partial differential equations (PDE) by generating a hierarchy of successively coarser discretizations and recursively using the solution on a coarser grid to update the solution on a finer grid. The methods are attractive as they avoid the need for an expensive solver with quadratic cost on the fine discretization of the problem. Algebraic Multigrid (AMG) generalizes this concept to problems that do not originate as the discretization of a PDE but build up the hierarchy on the system matrix of the linear system. Because of its robustness and efficiency, AMG has become a central component of many scientific computing applications in academia and industry. With modern supercomputers increasingly incorporating GPU accelerators and low precision support, its popularity raises demand for redesigning to leverage the fine-grain parallelism of GPUs and employ mixed precision strategies to reduce the runtime and memory footprint.

In this dissertation, we design and implement the first open-source high-performance AMG implementation that allows users to choose the precision format used in the distinct grid levels individually while providing platform portability across GPUs from AMD, Intel, and NVIDIA. We note that the development of this algorithm is heavily intertwined with the development of the Ginkgo open-source software library. For this reason, the dissertation also includes significant detail on how the mixed precision AMG algorithm influenced and extended the design and capabilities of the Ginkgo library. We explain how we extended the scope of Ginkgo from supporting NVIDIA GPUs to supporting GPUs from AMD and Intel. We discuss how the sparse matrix vector product (SpMV) is the backbone of many sparse applications and demonstrate that optimizing this kernel improves AMG performance immediately. We show that the developed high-performance AMG embraces flexibility in terms of AMG options and portability in terms of supporting different hardware platforms while remaining competitive with vendor libraries like NVIDIA's AmgX implementation. Finally, we introduce the idea of using lower precision formats for subsequent matrices to enhance the performance and memory footprint. We use experiments on real-world applications to showcase the numerical challenges that can arise and discuss problem-specific algorithmic strategies to overcome these. In performance experiments, we demonstrate that using low precision or a hierarchy of lower precision formats can reduce the overall execution time when using AMG in production for real-world problems.

Kurzfassung

Merhgittermethoden sind Algorithmen zur Lösung von partiellen Differenzialgleichungen (PDE), die eine Hierarchie von sukzessive gröber werdenden Diskretisierungen generieren und anschließend die Lösung eines gröberen Gitters rekursiv benutzen, um die Lösung eines feineren Gitters zu aktualisieren. Die Methoden sind attraktiv, da sie keine teuren Löser, deren Kosten quadratisch mit der Größe der Diskretisierung steigt, benötigen. Algebraische Mehrgittermethoden (AMG) erweitern dieses Konzept zu Problemen, die nicht von der Diskretisierung einer PDE stammen. Dabei wird die Hierarchie alleine aufgrund der Systemmatrix gebildet. Durch die Robustheit und Effizienz der AMG ist diese Methode zu einer Schlüsselkomponente für viele Anwendungen des wissenschaftlichen Rechnens in der Forschung und Industrie. Mit der steigenden Benutzung von GPU-Beschleunigern und verringerter Genauigkeit auf modernen Supercomputern steigt auch der Bedarf AMG neu zu designen, um die massive Parallelität der GPUs auszunutzen und ihre Laufzeit und Speicherbedarf mithilfe von gemischter Genauigkeit zu verringern.

In dieser Dissertation designen und implementieren wir die erste Open-Source high-performance AMG, die es Nutzern erlaubt, die Präzision für jedes einzelne Hierarchie-Level zu bestimmen, und gleichzeitig portabel auf GPUs von AMD, Intel, und NVIDIA eingesetzt werden kann. Diese Entwicklung ist eng mit der Open-Source Bibliothek Ginkgo verknüpft. Daher enthält diese Dissertation auch wichtige Details, wie die gemischte Präzision AMG das Design und die Fähigkeiten von Ginkgo beeinflusst und erweitert hat. Wir erklären, wie wir Ginkgo erweitert haben, neben NVIDIA GPUs auch AMD und Intel GPUs zu unterstützen. Wir beschreiben die Wichtigkeit des Matrix-Vektor-Produkts für dünnbesetzte Matrizen (SpMV) für viele wissenschaftliche Anwendungen und zeigen, dass die Optimierung dieses Kernels, die Performanz des AMG direkt steigert. Wir zeigen die Flexibilität der entwickelte high-performance AMG Implementierung in Bezug auf Methoden-bezogene Optionen und Portabilität auf verschiedene Hardware-Plattformen. Gleichzeitig bleibt die Implementierung kompetitiv mit Herstellerbibliotheken wie NVIDIAs AmgX. Zum Schluss führen wir Matrixformate mit verringerter Präzision ein auf tieferen Hierarchie-Level ein, um die Performanz und Speicherplatzbedarf zu verbessern. Mittels Praxis-relevanten Anwendungen werden mögliche numerischen Herausforderungen dargestellt und wir diskutieren problemspezifische Lösungsstrategien. In Performanz-Experimenten zeigen wir, dass die hierarchische Benutzung von niedriger Präzision innerhalb der AMG die gesamte Laufzeit zur Lösung relevanter Probleme verringern kann.

Acknowledgements

I want to send my deepest thanks to Prof. Hartwig Anzt, who has offered me the opportunity to work in the Ginkgo team and Fixed-Point Methods for Numerics at Exascale (FiNE) in Karlsruhe Institute of Technology (KIT). Despite studying in Germany, I improved my English skills significantly instead. He encouraged me to try new ideas and supported me to collaborate internationally. He introduced my work to many talented researchers, which resulted in several impactful projects. He also builds up a friendly and high-efficiency group. I enjoy working on the challenging Ph.D. research with the team.

I want to thank all the people in the Ginkgo team. Without collaborating with the team, it is impossible to finish this work. We put a lot of effort into improving the library's sustainability, correctness, and usability. This sustainable software workflow significantly impacts how I work on software packages. I will keep this habit in my future career. Dr. Terry Cojean, Dr. Yen-Chen Chen, and Pratik Nayak encouraged me to take this opportunity for a Ph.D. Without the summer visit to KIT, my plan was never to continue an academic career in a Ph.D. direction. I am also grateful to Terry, Pratik, and Thomas Grützmacher. In the beginning, We exchanged many ideas, solved several issues, and figured out some projects in the four-person office. Terry advised me a lot about paper writing and many concerns that need to be taken care of in the code. I got help from Thomas and Pratik to handle the paperwork and communication correctly in Germany. They also helped me understand how the Ph.D. program works and assisted me throughout the application process. We have discussed a lot about the designs, concerns, and implementation in the Ginkgo team. Dr. Marcel Koch and Tobias Ribizel usually bring different viewpoints and concerns such that I always notice something I have never thought about. Sometimes, we have come up with new ideas from the discussions. I am proud to be a member of the Ginkgo team.

Karlsruhe, December 2023 Yu-Hsiang Tsai

Funding and Resource Acknowledgements

This work was supported by the “Impuls und Vernetzungsfond” of the Helmholtz Association under grant VH-NG-1241, and the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Part of this work was performed on the bwUniCluster supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and the Universities of the State of Baden-Württemberg. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Part of this work 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.

Part of this work used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction and Motivation

With the success of deep learning and the importance of high-performance computing (HPC), the hardware landscape is changing, with more and more general-purpose graphics processing units (GPUs) being integrated into the leadership supercomputers in the USA and Europe. AMD, Intel, and NVIDIA are the leading vendors providing GPUs and the corresponding ecosystem. The Exascale computing project (ECP) in the USA setups three HPC systems from three vendors: Summit with NVIDIA V100, Frontier with AMD MI250X, and Aurora with Intel Data Center GPU Max 1550 (Intel Max1550). HPC systems equipped with different GPUs and programming ecosystems are not only presented in the US but also in Europe. The European High-Performance Computing Joint Undertaking (EuroHPC JU) also builds LUMI with AMD MI250X and Leonardo with NVIDIA A100. Leibniz-Rechenzentrum installs the Intel Max1550 on the SuperMUC-NG supercomputer. This trend hints that software libraries should consider portability on different GPUs with different ecosystems.

Mixed precision is an increasingly important area in deep learning and high-performance computing. Applications and hardware use low-precision computations to improve performance. Sometimes, we can use low precision directly because the problem only requires low accuracy. However, we usually aim for high accuracy in the final output but use the low precision to accelerate the performance. As operations on sparse matrices are usually memory-bound, a lot of research focused on reducing communication and data access. Mixed precision also follows this idea by partially reducing the data size. Without a doubt, changing everything to low precision can improve performance. However, we can not achieve the same accuracy as high precision. While algorithms exist that base the selection of the precision formats on pre-defined accuracy, most mixed precision algorithms aim to preserve high precision while using low precision for faster execution.

Scientific applications need to change their design and algorithms to be able to run on different vendors' GPUs and improve performance from mixed precision. Adapting to different GPUs requires to optimize the algorithms for different architectures. It will be more challenging if applications want to use the native software ecosystems from vendors because it may require reimplementing the kernels in quite different ways. When using a portability language to implement algorithms, applications may need to allow some additional overhead or lack of cutting-edge features. Building algorithms with mixed precision is another challenge. The mixed precision algorithm must reach the same high accuracy as the original algorithm. Additionally, applications may face the lack of mixed precision routines from math libraries, such that algorithms require additional workaround or implement the high-performance math routines. This additional workaround may lead to the mixed precision implementation being worse than the original implementation. Thus, the mixed-precision and high-performance math libraries on these GPUs are essential for these scientific applications.

Algebraic Multigrid (AMG) has received considerable attention from many applications. MFEM [And+21; Mfe] is a scalable C++ library for finite element methods. It supports arbitrary high-order finite element meshes/spaces and a wide variety of finite element discretization approaches on GPU. MFEM uses AMG to solve 2D/3D diffusion problems, elasticity eigenproblems, Laplacian problems, and topology optimization problems. OpenCARP [ope+23; Pla+21] is an open-source cardiac electrophysiology simulator. The openCARP bidomain implementation relies on AMG's performance for the elliptic solve component of the electrics computations. NekRS [Fis+22] is a fast and scalable computational fluid dynamics (CFD) focusing on high-fidelity turbulence simulation and thermal fluid simulation. NekRS uses AMG as the coarsest solver of the p-multigrid solver. AMG performance and portability are important for these applications.

Based on the above observations, we propose to design and implement a portable mixed-precision algebraic multigrid on high-performance GPU with optimized and flexible kernels and SpMV. In this dissertation, we develop the first open-source and platform portable AMG solver that can also use mixed precision. We heavily rely on the sustainable, open-source, and high-performance sparse linear algebra library - Ginkgo. To achieve platform portability, we extend Ginkgo backends from NVIDIA GPUs to AMD and Intel GPUs. Thanks to Ginkgo's design, we can use the native software ecosystem to achieve high performance. We develop competitive sparse matrix vector multiplication (SpMV) and extend the support for comprehensive mixed precision SpMV, which is one of the backbones of the AMG algorithm. We design a composable AMG library, which can reuse the existing Ginkgo's solvers and preconditioners. The implementation based on our design is competitive with the state-of-the-art AMG library - NVIDIA's AmgX. Our AMG design gives users the possibility to set the numerical precision flexibly, thereby enabling the mixed-precision AMG. Using low precision in the lower levels of a multigrid method can accelerate the computations without impacting the accuracy of the final result.

This dissertation is composed of six themed chapters. Chapter 2 introduces the background and the necessary information for the dissertation's topics. Chapter 3 illustrates the sustainable design of Ginkgo, the portability challenges, and our solution. Chapter 4 shows the competitive implementation and mixed-precision capabilities of Ginkgo's SpMV. Chapter 5 introduces Ginkgo's AMG design and demonstrates it being competitive in performance with the state-of-the-art NVIDIA's AmgX due to high-performance SpMV in Chapter 4 and low overhead AMG design. Chapter 6 contains the design of mixed-precision AMG and the performance improvements coming with the use of low precision. Thanks to flexible SpMV in Chapter 4 and our AMG design in Chapter 5, we also check several mixed-precision configurations in Chapter 6. With the portability structure from Chapter 3, all components in Chapters 4 to 6 are available on AMD, Intel, and NVIDIA GPUs. Finally, we conclude our ideas, observations, and experiments in Chapter 7.

The main contributions of this dissertation are as follows:

- Extend Ginkgo support from NVIDIA GPUs to AMD/NVIDIA/Intel GPUs.
- Design and implement high-performance SpMV competitive with vendors' libraries.
- Extend high-performance SpMV to support mixed precision.

- Design a portable and highly flexible AMG that is competitive with the state-of-the-art NVIDIA's AmgX under the same double-precision configuration.
- Extend the design to mixed-precision AMG that outperforms the double-precision AMG.
- Analyze the convergence and performance implications of mixed-precision AMG in comparison with standard AMG.

The content of this dissertation is largely based on the following first-authored publications:

- Sparse Linear Algebra on AMD and NVIDIA GPUs - The Race Is On [TCA20]
- Preparing Ginkgo for AMD GPUs - A Testimonial on Porting CUDA Code to HIP [Tsa+21]
- Porting Sparse Linear Algebra to Intel GPUs [TCA22]
- Providing Performance Portable Numerics for Intel GPUs [TCA23]
- Mixed Precision Algebraic Multigrid on GPUs [TBA23a]
- Three-precision Algebraic Multigrid on GPUs [TBA23b]

2. Background

We want to solve a linear system $Ax = b$ with the matrix A and the right-hand side from applications on high-performance GPUs. We go through the architecture and their ecosystems from AMD, Intel, and NVIDIA in Section 2.1 to raise the need for a portable math library in Chapter 3 due to the inconsistency among vendors. With the roofline model in Section 2.2, we observe that SpMV and SpMV-centric algorithms (like AMG and iterative solvers) are memory-bound kernels. Reducing the memory footprint can improve the performance, which inspires the mixed-precision design to improve the performance of AMG in Chapter 6. We introduce the different precision format in Section 2.3 and the sparse matrix format in Section 2.4 such that we can efficiently store the matrix in a particular format with a certain precision. Section 2.3 also indicates the accuracy and range from different precision formats, which affect the accuracy of solvers. Hence, the mixed-precision design also needs to consider the effect of lower precision operations. We finally introduce Multigrid algorithms in Section 2.5, and Krylov solvers and the iterative update scheme in Section 2.6 for latter Chapters 5 and 6.

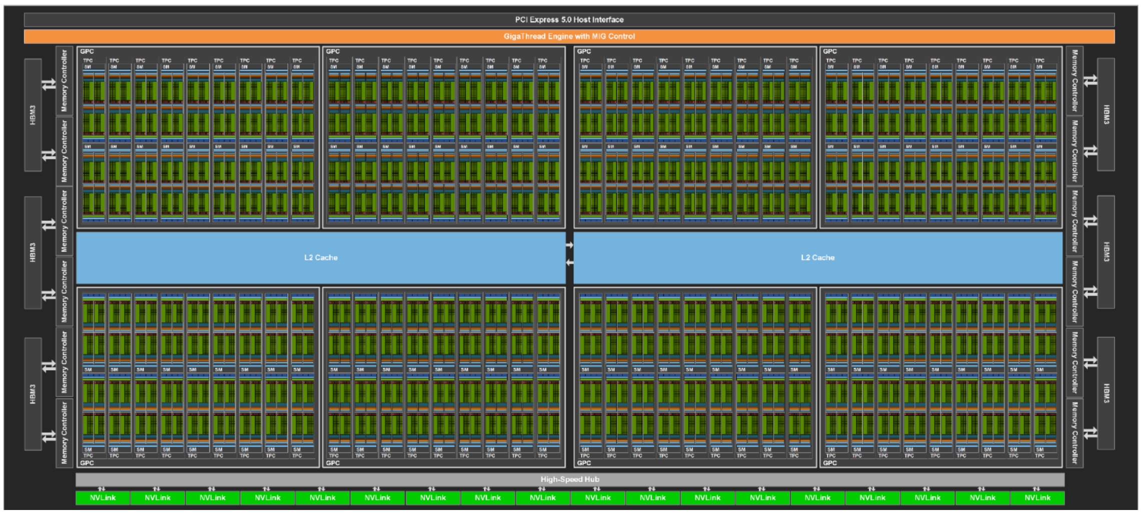


Figure 2.1: H100 GPU architecture figure from NVIDIA whitepaper [Nvia]

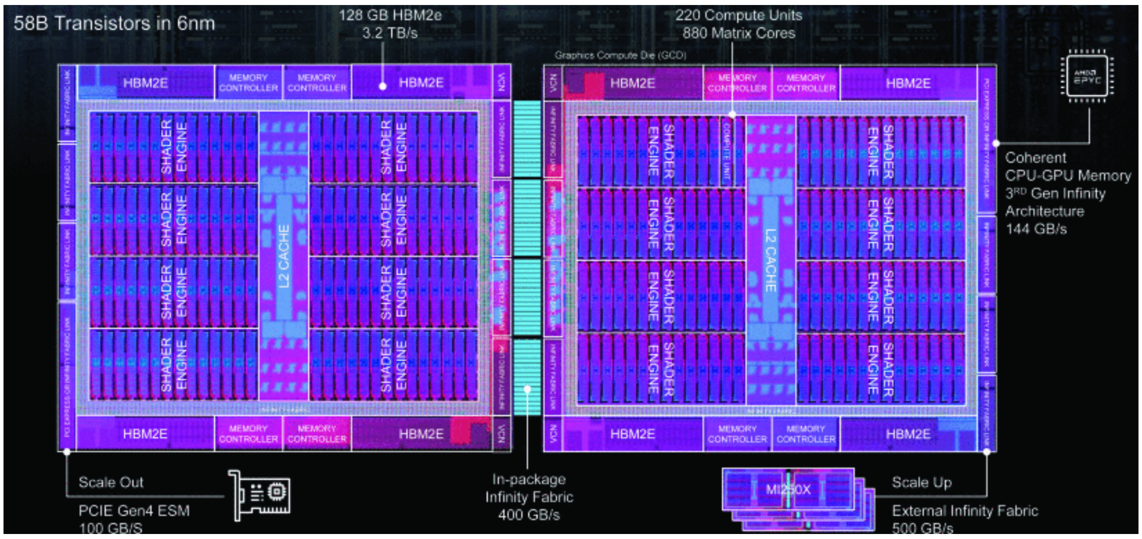


Figure 2.2: MI250X GPU architecture figure from AMD Hot Chip presentation [SJ22]

2.1 Heterogeneous Architectures and Software Ecosystems

Modern supercomputers often feature GPU accelerators. As of today, we primarily see GPUs deployed from NVIDIA, AMD, and Intel. While the hardware designs share some similarities (see Figures 2.1 to 2.3), the vendors do not share a common programming ecosystem. Instead, the different GPUs come with their hardware-native programming model: NVIDIA GPUs with CUDA, AMD GPUs with ROCm/HIP, and Intel GPUs with SYCL. Although some programming languages aim for several vendors' devices, they can only provide the common part among these devices and do not provide the excellent performance as the vendors' languages. This poses a challenge to software developers that aim at platform portability.

The vendors do not only use different programming models, but also different terms for their programming concepts, even though these are often very similar. We provide an overview of technical terms in Table 2.1. In general, this dissertation uses

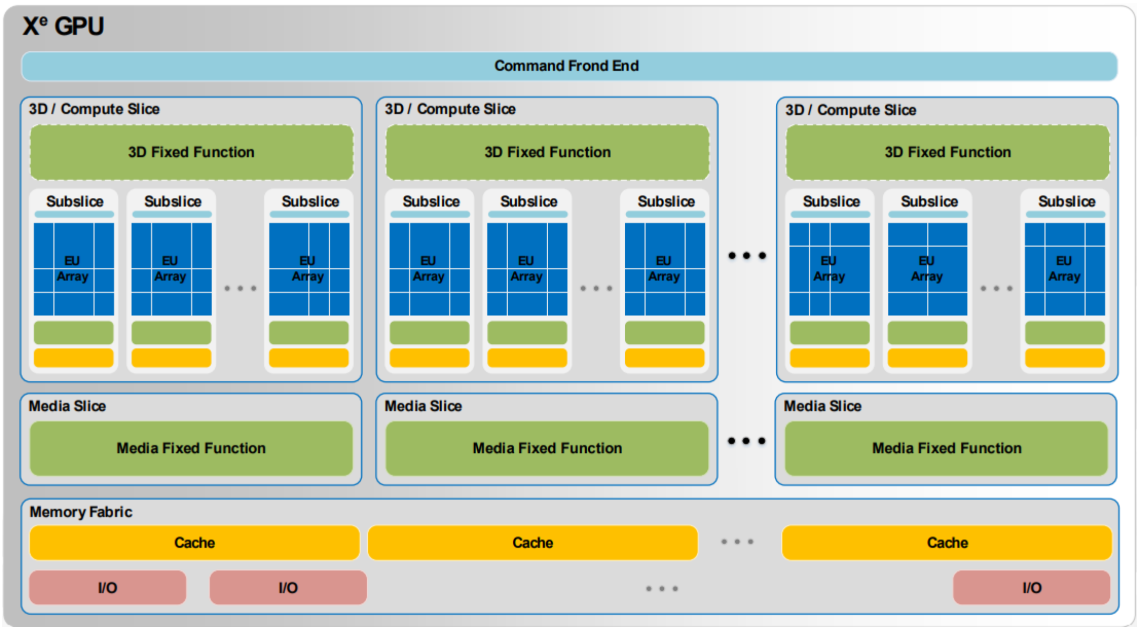


Figure 2.3: Xe GPU architecture figure from Intel Hot Chip presentation [Bly20]. Intel Max1550 packs two tiles together.

CUDA	HIP	SYCL
thread	thread	work-item
block	block	work-group
warp	wavefront	sub-group
subwarp	subwavefront	no[1]
register	register	private memory
shared memory	shared memory	local memory

Table 2.1: The technical terms in different vendors' languages.
[1]Intel extensions do not officially support it yet. In this dissertation, we may use sub-sub-group as a virtual concept.

the CUDA-style terms when discussing the concepts. We use specific terms when only discussing a specific ecosystem. SYCL currently does not allow sub-sub-group operations like CUDA/HIP, so we must have the same conditions and sub-group operation for an entire sub-group. We use the term sub-sub-group only for the algorithm description and ensure all threads in the same sub-group execute the same instruction in SYCL.

The different vendors' GPUs share a hierarchical programming model, but how memory is handled differs. We visualize the common memory model in Figure 2.4. The model describes the hierarchy of memory and parallelism. Developers try accessing the low-latency memory more than the high-latency memory to improve the kernel performance. The hierarchy of memory also shows the hierarchy of memory latency: (low) registers < shared memory < global memory (high). The size of memory is also: (small) registers < shared memory < global memory (large). Threads own their registers which are not shared with the other threads. If the threads are part of the same warp (or subwarp), they can exchange the register data with some specific hardware instruction. Subwarp sizes are restricted to powers of 2, and it

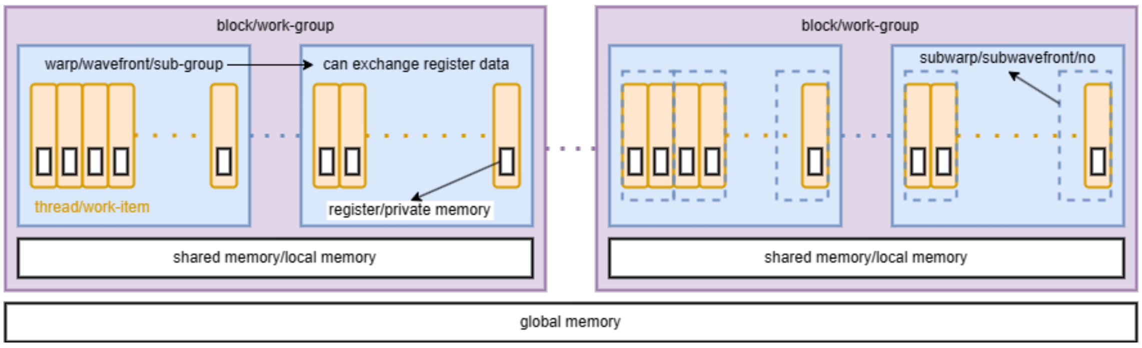


Figure 2.4: The common memory model among AMD, Intel, and NVIDIA GPUs. We label the items with their own technical terms. One term: all use the same name; two terms: AMD/NVIDIA use the first name, but SYCL uses the second one; three terms: NVIDIA uses the first name, AMD uses the second one, and SYCL uses the last one.

indicates the operations only in a small range, not the whole warp. All threads in the same block have access to the same shared memory such that they can share or exchange data via the shared memory. Threads part of one thread block can not access the shared memory part of another thread block. However, threads of distinct thread blocks can exchange information over the global device memory. There is a cache to fetch the data from shared memory and global memory, so programmers need to synchronize correctly (block synchronization for shared memory or device synchronization for global memory) or use hardware instruction to ensure that the data is not outdated. Note that we do not account for the thread block clusters model and distributed shared memory from NVIDIA Hopper architecture[Nvib] in this model and this dissertation.

To abstract the memory control and synchronization across levels of the hardware hierarchy, CUDA 9 introduced cooperative groups in [Coo]. For example, it provides the same interface for subwarp with different sizes and entire warp to exchange the register data. Also, the synchronization functions are the same across different levels. We do not need to worry about passing the wrong mask to the warp synchronization function. The abstraction is handy for programming and allows the reuse of the same code for different subwarp sizes. HIP added a similar interface after 4.5. SYCL does not have a similar interface for this abstraction. We show a workaround to minimize the gap between the ecosystems in Chapter 3.

Kernel launches differ in the different programming models. In Listing 2.1 and Listing 2.2, we compare the kernel launch in CUDA (HIP) with the kernel launch in SYCL. CUDA launches kernels with the <<<>>> syntax: grid size, block size, (optional) dynamic shared memory size, and (optional) stream. The memory ordering follows the first index first (blockx). CUDA allocates the static memory in the kernel, so it is not shown in this example. HIP has the same launching style as CUDA. SYCL uses a queue-based kernel launch syntax. A SYCL queue is similar to a CUDA stream if we enable the in_order property. SYCL needs to allocate the shared memory out of the kernel no matter if it is static[1] or dynamic. Another big difference is that the memory order uses the last axis first. That means, we need to reverse the order to launch a thread configuration identical to the CUDA/HIP configuration. SYCL only

[1]Intel provides an extension to allocate static shared memory in the kernel.

needs the number of total threads (grid × block in the example), not the number of blocks(work-groups). We discuss the difference in more detail in terms of kernel aspects in Chapter 3. Understanding these differences in hardware and programming language syntax and concepts is essential to addressing the portability challenges.

```
// CUDA launch kernel
cuda_kernel<<<dim3(gridx, gridy, gridz), dim3(blockx, blocky, blockz),
    dynamic_memory_size, stream>>>(...);
```

Listing 2.1: Launch CUDA kernel

```
// SYCL launch kernel
queue->submit([&](sycl::handler& cgh) {
    sycl::accessor<....> dynamic_shared_memory(cgh);
    sycl::accessor<....> static_shared_memory(cgh);
    cgh.parallel_for(
        sycl::nd_range<3>(range<3>(gridz, gridy, gridx) * range<3>(blockz, blocky,
            blockx),
                          range<3>(blockz, blocky, blockx) /* work-group size */),
        [=](sycl::nd_item<3> item_ct1) {
                sycl_kernel(..., item_ct1, dynamic_shared_memory,
                    static_shared_memory);
            });
});
```

Listing 2.2: Launch SYCL kernel

2.2 Roofline Model

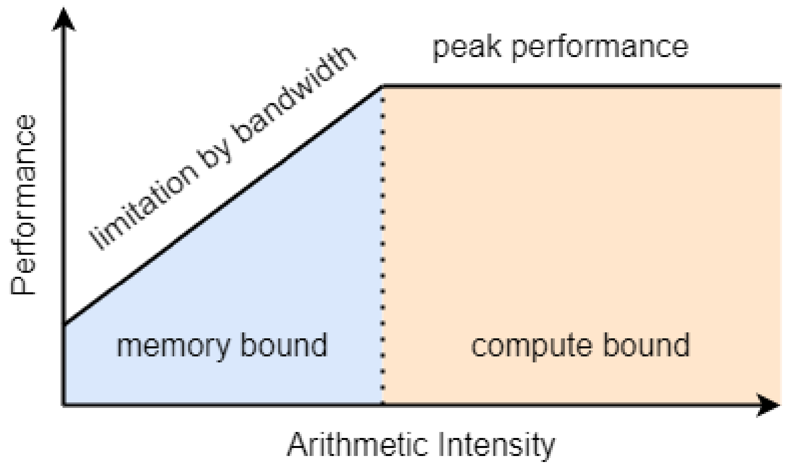


Figure 2.5: The roofline model.

The roofline model introduced in [WWP09] is a performance model to identify the performance bound of kernels on specific architectures. The roofline model needs to know the **Arithmetic Intensity** (or operational intensity) of the kernel, which is computed by the number of operations and the memory traffic. The formula is

$$\textbf{Arithmetic Intensity} = \frac{\textbf{number of operations}}{\textbf{memory traffic}}$$

We can measure the peak bandwidth and peak performance using a mini app such as BabelStream[Eva] and mixbench[KC17], or compute the theoretical bounds from hardware's characteristics. With the peak performance and peak bandwidth, we can get the performance formula with the given Arithmetic Intensity:

$$\text{Performance} = \min\left(\text{Peak Performance}, \text{Peak Bandwidth} \times \text{Arithmetic Intensity}\right)$$

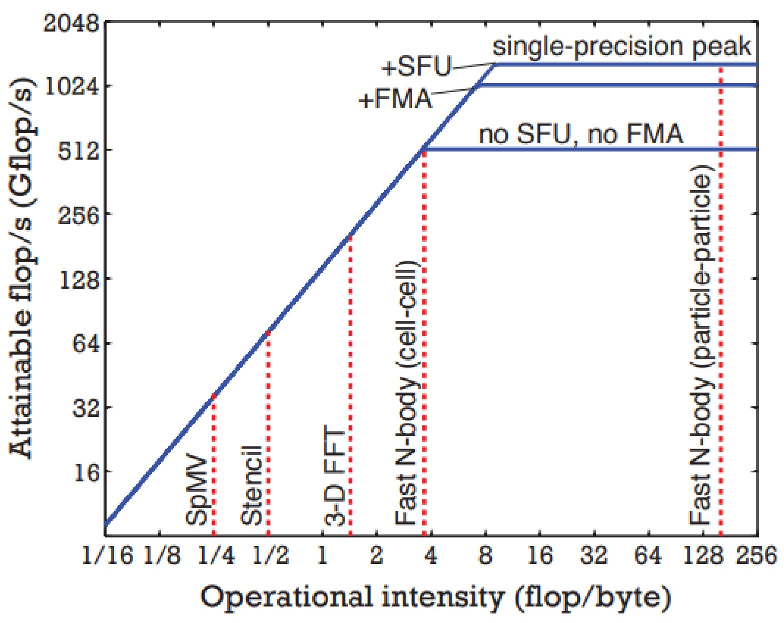


Figure 2.6: The roofline model with kernels of fast multipole method from [YB12] on NVIDIA C2050 GPU. SFU is the special function unit and FMA is the fused multiply-add instruction. The order of multipole expansions was set to 15.

and plot the figure like Figure 2.5. With the roofline model, we can put the kernels into two categories: memory-bound (Arithmetic Intensity $\leq \frac{\text{Peak Performance}}{\text{Peak Bandwidth}}$) or compute-bound (Arithmetic Intensity $\geq \frac{\text{Peak Performance}}{\text{Peak Bandwidth}}$) kernels. $\frac{\text{Peak Performance}}{\text{Peak Bandwidth}}$ is called the machine balance, which is a cut-off point for compute-/memory-bound kernels. On the left side of Figure 2.5, the memory bandwidth limits the performance of the kernels. On the right side, the architectures' operations performance limits the performance of kernels. As Section 2.1 mentioned, current accelerators follow a hierarchical design that contains several layers for memory and particular computing units, so some papers such as [CP14; IPS14] extend the roofline model with more bound from different layers of the modern devices.

Sparse routines and Basic Linear Algebra Subprograms (BLAS) 1/2 routines, which do dense vector-vector or matrix-vector operations, are usually memory-bound, but BLAS 3 routines for dense matrix-matrix operations are compute-bound kernels. As an example, Rio and Lorena [YB12] collect the Arithmetic Intensity of several kernels of the fast multipole method in Figure 2.6. In addition, Figure 2.7 shows that the trend of machine balance tends to increase in the future. With the roofline model, we know that we should reduce the memory traffic for memory-bound kernels but reduce the operations for compute-bound kernels to improve the kernel performance. The mixed-precision idea is to use a lower precision format (Section 2.3) to reduce the memory traffic. Because the lower precision will affect the accuracy, mixed precision also needs to take the accuracy into account.

2.3 Precision Format Scheme

In computational mathematics, numbers are represented in a machine-internal floating point format of limited accuracy. We need to select the precision to store our data in the matrices Section 2.4 and vectors. The precision format affects the range

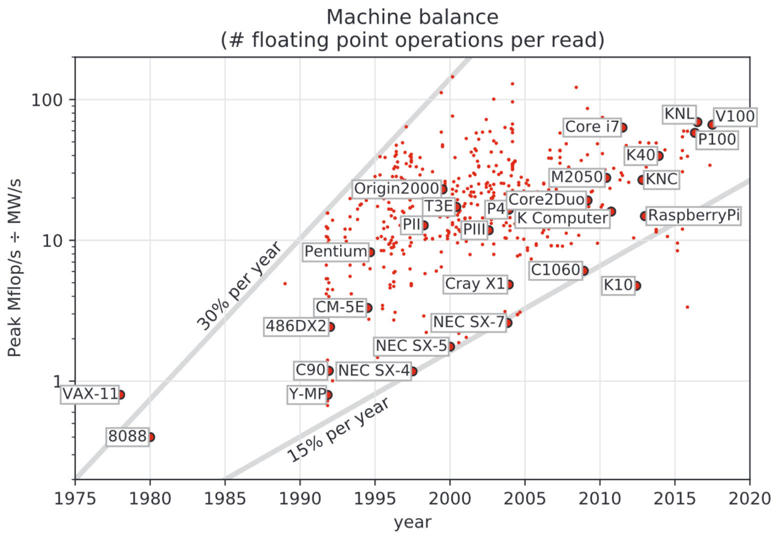


Figure 2.7: The trend of machine balance from [Abd+21]

of representable values and the solvers' accuracy, making the mixed-precision a challenging topic in Chapter 6. The common (IEEE standard-like) floating point is composed of three segments:

- **Sign bit**: The sign of the number.
- **Exponent**: Exponent field stores the exponent value in offset-binary representation. The actual exponent value for the number is $stored\_value - (2^{bitsizeof(Exponent)-1} - 1)$, where $bitsizeof(Exponent)$ gives the number of bits for the exponent field. $2^{bitsizeof(Exponent)-1} - 1$ is also called exponent bias. The bias makes the unsigned integer can represent a negative value. Two precision formats have the same range if they have the same number of bits for exponent.
- **Significand precision(fraction)**: The value represents the significand decimal digits. It assumes the leading digit before the decimal point is 1 for a normal number.

By combining these three segments of bit representation, we can have the following formula for floating point.

$$(-1)^{sign}(1.fraction)_2 \times 2^{exponent-bias}$$

We collect the three segments for the representation of formats in Table 2.2. There are two most common floating point formats from the IEEE 754 standard [Iee]: double precision uses 64 bits (DP) and single precision uses 32 bits (SP), which most compilers support. There are two 16-bit floating point number representations: half precision (HF) from IEEE 754 standard and bfloat16 precision (BF) introduced in [Dil+17] from Google Brain. The 16-bit formats are gaining popularity due to their use in artificial intelligence (machine learning). Some precision formats do not use multiple times of 2 bytes (16 bits) as size, such as NVIDIA's TensorFloat introduced in [Kha20] with A100 release uses 18 bits.

There are some special meanings for specific exponent encoding.

	total bits	sign bit	exponent	fraction	IEEE standard	machine epsilon
double(DP)	64	1	11	52	Yes	2^{-52}
single(SP)	32	1	8	23	Yes	2^{-23}
half(HF)	16	1	5	10	Yes	2^{-10}
bfloat16(BF)	16	1	8	7	No	2^{-7}

Table 2.2: Different precision formats for floating point number

- **All zeros in exponent**: It is zero if the fraction part is also all zeros. Otherwise, it represents the subnormal numbers.
- **All ones in exponent**: It represents the infinite(Inf) if the fraction part is all zeros. Otherwise, it represents the Not-a-Number(NaN).

The subnormal number fills the underflow gap between the smallest normal number and zero and prevents underflows when using addition and subtraction on two floating point numbers. The subnormal number representation is

$$(-1)^{sign}(0.fraction)_2 \times 2^{1-bias}$$

2.3.1 Rounding Error in Floating Point Formats

We use the widespread variant definition for machine epsilon ($\epsilon_{machine}$): machine epsilon is the difference between 1 and the next larger floating point number. That is $2^{-bitsizeof(fraction)}$ in Table 2.2. This definition is widely used in programming languages (such as C++, Python, and Rust), scientific software (such as MATLAB and Mathematica), and papers [Hig02; QSS07], which is slightly different from the formula in [And+99; Dem]. As we can not always store the value in the floating point representation exactly, the truncation error represents the difference between the stored value and the actual value. There are two special cases, which are usually not considered in the numerical analysis, in the truncation error: underflow - the value is too small such that it is zero in the precision format, and overflow - the value is too large such that it is infinite in the precision format. We use $fl : \mathbb{R}- > F$, F is the set of represented values in a given floating-point representation. That is, $v = fl(x)$ gives the actual stored value v when storing x in the representation. We have the following formula:

$$fl(x) = x(1+\epsilon), \text{ which } |\epsilon| \leq \epsilon_{machine}$$

Besides the truncation error, we also have an error from the operation by using floating point schemes. The rounding error represents the difference between the result produced by a given algorithm using exact arithmetic and floating-point arithmetic. Ideally, the floating-point operation (FLOP) produces the correctly rounded result, i.e., x flop $y = fl(x \text{ op } y)$, where flop is the floating-point version of arithmetic operation op. For example, the IEEE standard achieves this ideal as long as x op y is within the range of a floating-point system. We have the standard model for the rounding error [Hea18]

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\epsilon), \text{ which } |\epsilon| \leq \epsilon_{machine}$$

The error from the precision format is bound by the machine's epsilon. Also, there is a heuristic rule for the solver accuracy from [Wat04] on the solution x' by solving the linear system $Ax = b$ in

$$\frac{\|x' - x\|}{\|x\|} \approx \epsilon_{machine} \times cond(A)$$

$cond(A)$ is the conditioner number of the matrix A, and it is always larger than or equal to 1. The precision formats have different machine epsilon as in Table 2.2. Hence, the precision format selection is important for accuracy. Suppose we require an accuracy that is less than the machine epsilon of single precision. In that case, we can not use single precision throughout an algorithm but need to design, for example, a smart algorithm that combines single precision computations with a double precision outer loop to retrieve high-precision output.

2.4 Sparse Matrix Formats

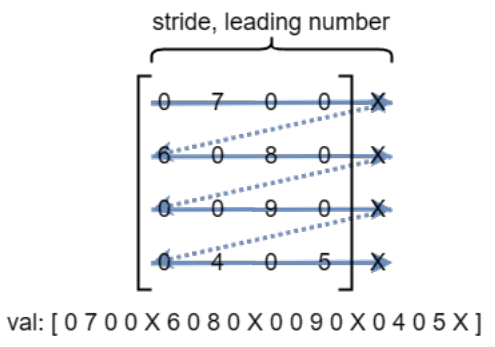
A simple way of storing a matrix is the DENSESection 2.4.1 format storing all matrix entries in row- or column-major order in an array. However, if a matrix contains mostly zeros, a more efficient strategy is to store only the nonzero values explicitly. We call the matrices containing many zeros "sparse matrices". For sparse matrices, we can design sparse data formats, such as only storing the nonzeros, that reduce the memory footprint and the computational cost of operations involving the matrix. [Bar+94]. According to processors' parallel design, some matrix formats also store unused elements as padding to improve memory access and data parallel efficiency like ELL [BG09]. In the following, we describe the following matrix formats as we will use these in the Chapter 4. We describe the main formats DENSE, CSR, COO, ELL, and HYBRID with the example of Figure 2.8. We also calculate the memory consumption on an $m \times n$ matrix with nnz nonzeros. All indices use 0-based indexing in the following description.

$$\begin{bmatrix} 0 & 7 & 0 & 0 \\ 6 & 0 & 8 & 0 \\ 0 & 0 & 9 & 0 \\ 0 & 4 & 0 & 5 \end{bmatrix}$$

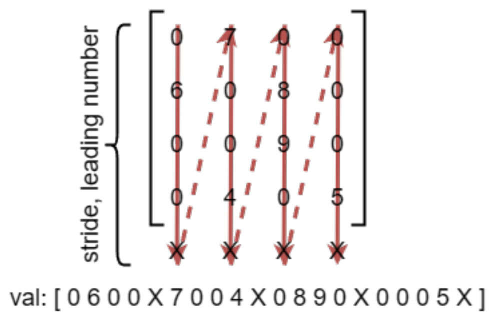
Figure 2.8: A 4 x 4 matrix with 6 nonzeros example.

2.4.1 DENSE

DENSE stores all values of the matrix explicitly. Two popular data-storage layouts to store the data are row-major and column-major. The row-major way stores the data row by row, that is, storing the first row and then the next row. The column-major way stores the data column by column. Sometimes, it stores the data with padding by stride or leading dimension number, which fills the matrix with unused data until it reaches a certain number before storing the next row in row-major (or next column in column-major). The padding can be used for



(a) DENSE matrix format stored in row-major fashion.



(b) DENSE matrix format stored in column-major fashion.

Figure 2.9: DENSE matrix format.

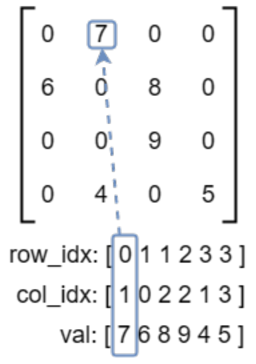


Figure 2.10: COO format stores the row index, column index, and value for each nonzero.

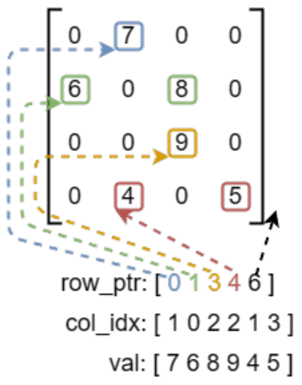


Figure 2.11: CSR format stores the column index and value for each nonzero, and the row pointer as the row offsets.

coalesced memory access or packing several matrices together. We use the example to visualize the row-major method in Figure 2.9a and the column-major method in Figure 2.9b. They need $m \times stride \times sizeof(value)$ bytes in the row-major method or $stride \times n \times sizeof(value)$ bytes in the column-major method. Storing the example in DENSE format requires $4 \times 5 \times 8 = 160$ bytes (without stride: 128 bytes) with double precision values.

2.4.2 COO

The most straightforward idea is to store nonzeros by their coordinates (row index, column index, and value), which is the COO format as in Figure 2.10. For a general case, COO requires $nnz \times (sizeof(value) + 2 \times sizeof(index))$ bytes of storage. With double precision for value and 4-byte integer coordinates, this example needs $6 \times (8 + 2 \times 4) = 96$ bytes.

2.4.3 CSR

If we always sort the elements by row index like Figure 2.10, the row index in the same row is repeated. Thus, Compressed Sparse Row (CSR) compresses those row indices to the row pointer format, which is visualized in Figure 2.11. The row pointer gives

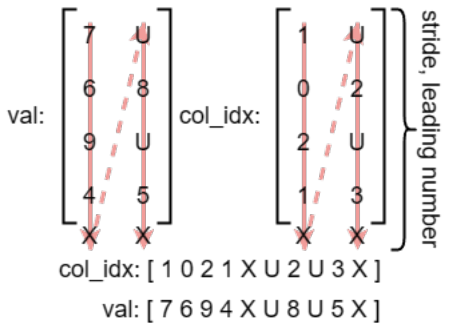


Figure 2.12: ELL format only stores column index, value for each nonzero by enforcing every row storage having the same number of elements

the index of the beginning of the row inclusively and the end of the row exclusively. The i-th row elements are in [row_ptr[i], row_ptr[i+1]) of the col_idx and val array. CSR also provides a more structured format than COO because CSR allows us to access the elements in certain rows in sequence. Because of the easy access to certain rows, algorithms often rely on the CSR format. In COO, we still need to search elements from a certain row even if the COO matrix is stored sorted. CSR is also called CRS (Compress Row Storage). We note that compared to COO, it is much more difficult to add nonzero elements to a CSR data structure. The memory formula is $nnz \times (sizeof(value) + sizeof(index)) + (m + 1) \times sizeof(index)$ bytes. With double precision for value and 4-byte integer, we need $6 \times (8 + 4) + (4 + 1) \times 4 = 92$ bytes for this example.

2.4.4 ELL

ELL (ellpack) uses padding in rows to enforce all rows to have the same number of stored elements, including the unused padding. Though this introduces overhead, it stores the data in column-major fashion such that the parallel resources can efficiently access and operate on the data in a SIMD fashion. Furthermore, the row-pointer is no longer needed as the number of stored columns reveals how many elements are stored in each row. In Figure 2.12, we need to fill the padding elements in rows (marked as “U”) if the rows have fewer elements than the given width of ELL (usually the longest row length). We need to distinguish “U” and stored elements from the storage, so we can not put anything like usual padding “X” case. We need to use some special value to mark it as not a value in a matrix. Although the “U” value depends on the implementation, “U” in col_idx usually uses -1 to mark the value as unused. There is a parameter stride or leading dimension number like DENSE. The memory usage is $stride \times width \times (sizeof(value) + sizeof(index))$ bytes. Figure 2.12 needs $5 \times 2 \times (8 + 4) = 120$ bytes (without stride: 96 bytes) when using double precision value and 4-byte integer to store the matrix in ELL format.

2.4.5 HYBRID

The HYBRID format tries to find a balance between ELL and COO. ELL is preferable for matrices whose rows contain the same number of elements. Thus, if the matrix is more balanced, it will be more suitable for ELL. The COO format handles the coordinates of elements without any compression or packing, so it will not be affected a lot by the matrix distribution. HYBRID splits the matrix to ELL and COO

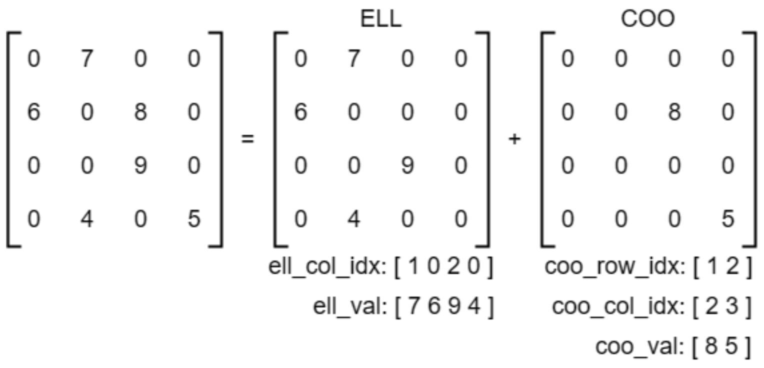


Figure 2.13: HYBRID format splits the matrix into two parts: one is stored in ELL, and the other is stored in COO. This example splits the first elements of each row into ELL and the rest into COO.

like Figure 2.13. It lets ELL handle the balanced part to achieve high parallel efficiency and leaves COO to handle the rest part of the matrix. Assuming i-th row contains nnz_i, the memory consumption of HYBRID is $ELL\_stride \times ELL\_width \times (sizeof(value) + sizeof(index)) + COO\_nnz \times (sizeof(value) + 2 \times sizeof(index))$, where $COO\_nnz = \sum (nnz_i - ELL\_width)^+$, whose $(*)^+$ only keeps positive values or zero for negative values. In the example Figure 2.13, it uses $4 \times 1 \times (8 + 4) + 2 \times (8 + 2 \times 4) = 80$ bytes with double precision value and 4-byte integer.

2.4.6 Summary

This section describes the most common formats - DENSE, COO, CSR, ELL, HYBRID. Even with a small example like Figure 2.8, the sparse matrix formats all use less memory than the DENSE format. We will focus on optimizing and implementing these formats later in Chapter 4. For completeness, we mention that there exist more formats like Sellp in [ATD14], SELL-c-σ in [Kre+14], or different kinds of hybrid formats. The formats are still increasingly being developed for specific applications or better performance due to the common usage of sparse matrix operations in scientific applications.

2.5 Multigrid

A multigrid method is an algorithm for solving differential equations using a hierarchy of discretizations. The main idea of multigrid is to accelerate the convergence of a basic iterative method (known as relaxation) by a global correction of the fine grid solution approximation from time to time, accomplished by solving a coarse problem. Some stationary iterative methods reduce the high-frequency error rapidly, but low-frequency error much more slowly. It causes a poor asymptotic rate of convergence. We call them smoother because they smooth the high-frequency error. Multigrid will generate the hierarchy from the matrix such that the low-frequency error of fine matrices is the high-frequency error of coarse matrices. Thus, when we apply the smoother to approximately solve the coarse matrices, smoothers can solve the high-frequency error rapidly on coarse matrices, which may be the low-frequencies error in the fine matrices. We recursively generate the hierarchy until a certain criterion such as the cost of direct solution in the coarsest grid is negligible compared to the cost of one relaxation sweep on the fine grid.

There are two Multigrid variants: geometric multigrid (GMG) and algebraic multigrid (AMG). A geometric multigrid derives the grid hierarchy from geometry information, i.e., the spatial discretization of the problem. An algebraic multigrid operates on the matrix representation of the problem and derives the hierarchy from the graph representation of the matrix. In both cases, a multigrid algorithm follows the steps outlined in Algorithm 1 to generate the hierarchy of a series of matrices. Different coarsening methods give different kinds of prolongation in Line 2 to lead different coarse matrices in Line 4. Falgout et al. [FS14] give a different way than the Galerkin product to generate the coarse matrix. The matrix generated from Algorithm 1 at the coarse level is smaller than the matrix at the fine level. There exist different coarsening strategies for AMG such as Parallel Maximum Independent Set introduced in [Lub85], Ruge-Stueben in [RS87], Pairwise Aggregation in [Not10], Parallel Modified Independent Set in [DSYH06], and Parallel Graph Match in [Nau+15].

Algorithm 1 Multigrid generation.

```
1: procedure MULTIGRID PREPARATION($F$)
2:     Define the prolongation(interpolation) $P$ from $F$ by the coarsening method.
3:     Define the restriction $R$ (usually $R = P^T$)
4:     Generate the Coarse Matrix $C = RFP$ (Galerkin product)
5:     if Reach the criterion for generation then
6:         Generate the coarsest solver on $C$
7:     else
8:         Generate the presmoother/postsmoother on $C$
9:         Continue Multigrid Preparation($C$)
10:    end if
11: end procedure
```

2.5.1 Multigrid Cycles

After hierarchy generation, multigrid cycles describe how to use the hierarchy to solve a problem. There are several kinds of cycles to use the hierarchy to solve the problem, like V-cycle in Algorithm 2 or W-cycle in Algorithm 3 with $\mu = 2$. Restriction and prolongation connect the information between the coarse level and the fine level. Restriction passes the residual ($= b - Ax$ in Line 7 in Algorithm 2) from the fine to the coarse level, and prolongation passes the solution update back to the fine level. (Pre/Post)-smoothers usually use less expensive operations to update the correction within a few iterations because we expect the smoothers solve the high-frequency error quickly. We visualize the computation components in Figure 2.14 on a two-level structure of multigrid. Figure 2.14 shows the important components with the icon: The circle icon (●) indicates the smoother for the solution update by smoothing the high-frequency error, the arrow-down icon (↘) indicates the restriction to the coarser level, and the level-up icon (↗) indicates the prolongation to the finer level. We reuse these icons in Figures 2.15a and 2.15b to describe the complete multigrid hierarchy for a V and a W cycle. The V-cycle only goes to the coarse level once in Figure 2.15a, but W-cycle uses the coarse level many times in Figure 2.15b. Figure 2.15a shows the different matrix sizes handled by different levels. How much the multigrid hierarchy decreases the matrix size depends on the coarsening strategy and the target problem.

Additionally to the V and W cycle, other multigrid cycles have been developed, such as the Full Multigrid V Cycle [BHM00] and the K-cycle [NV08].

Algorithm 2 V-cycle multigrid method.

```
 1: procedure VCYCLE($A$, $x$, $b$)
 2:     if it is the coarsest level then
 3:         Solve $Ax = b$ by the coarsest solver.
 4:         Return
 5:     end if
 6:     x = PreSmooth(x, b)
 7:     r = b - Ax
 8:     g = Restrict(r)
 9:     e = zero
10:     Vcycle(Coarse, e, g)
11:     x += Prolong(e)
12:     x = PostSmooth(x, b)
13: end procedure
```

Algorithm 3 μ-cycle multigrid method is a general representation from [BHM00]. $\mu = 1$ results in V-cycle like Algorithm 2 and $\mu = 2$ results in W-cycle

```
 1: procedure $\mu$-CYCLE($A$, $x$, $b$)
 2:     if it is the coarsest level then
 3:         Solve $Ax = b$ by the coarsest solver.
 4:         Return
 5:     end if
 6:     x = PreSmooth(x, b)
 7:     r = b - Ax
 8:     g = Restrict(r)
 9:     e = zero
10:     apply $\mu$-cycle(Coarse, e, g) $\mu$ times
11:     x += Prolong(e)
12:     x = PostSmooth(x, b)
13: end procedure
```

2.5.2 Coarsening Methods, Prolongation, and Restriction

Here, we discuss two coarsening strategies for algebraic multigrids, i.e., to generate a coarse matrix from a fine matrix. One is the classical-based method, which is inspired by the geometric multigrid. The point i **strongly depends** on the point j if

$$-A_{ij} \geq \theta \times \max_{k \neq i} \{-A_{ik}\}$$

with given a threshold value $0 < \theta \leq 1$. If the point i strongly depends on the point j, then the point j **strongly influences** the point i. Classical-based methods try to select the coarse point by the following two heuristic criteria from [BHM00]:

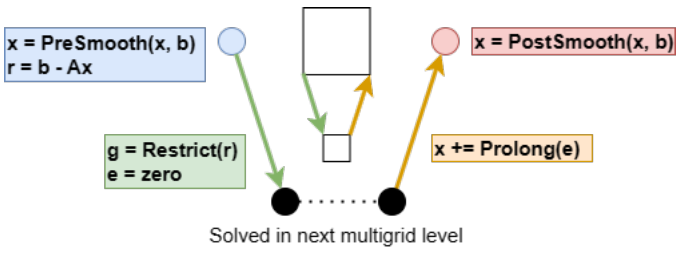
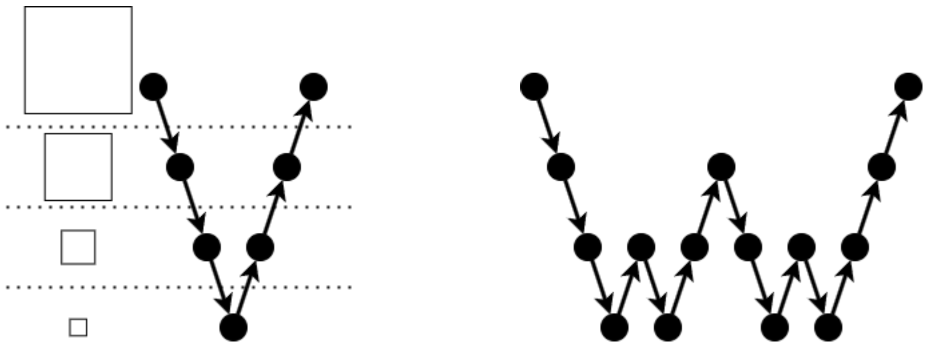


Figure 2.14: Computation components on the two-level structure of multigrid. Smoothers update the solution in the circle icon, restrictions pass the residual from the fine to the coarse level in the arrow-down icon, and prolongations pass the solution from the coarse to the fine level. Also, different levels have different system sizes, as shown in the square size in the middle.



(a) V-cycle multigrid visualization with the matrices size. V-cycle only visits the coarse levels once.

(b) W-cycle multigrid visualization. Unlike the V-cycle, it visits the coarse levels more often.

Figure 2.15: Multigrid cycle visualization.

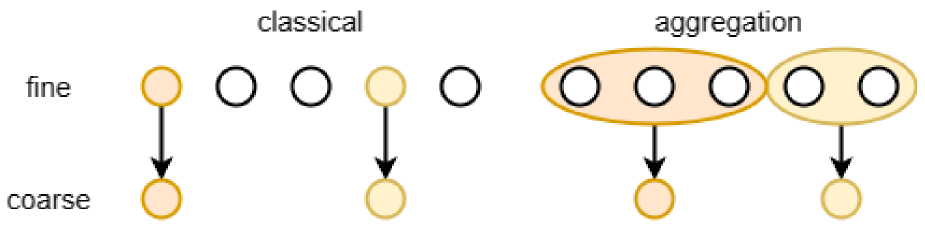


Figure 2.16: AMG generates coarse points by classical methods on the left side and aggregation-based method on the right side.

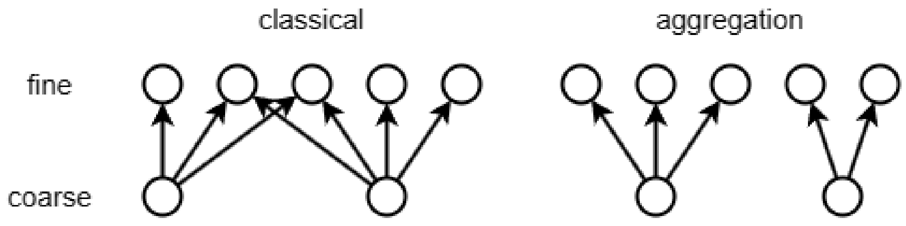


Figure 2.17: AMG generates prolongation by classical methods on the left side and (unsmoothed) aggregation-based method on the right side.

- **H-1**: For each fine point i, every point $j \in S_i$ that strongly influences i should be in the coarse interpolatory set C_i or should strongly depend on at least one point in C_i.
- **H-2**: The set of coarse points C should be a maximal subset of all points with the property that no C-point strongly depends on another C-point.

However, the algorithm may not always satisfy both **H-1** and **H-2**, so the classical method enforces **H-1** but uses **H-2** as a guide or vice versa. The classical method selects the coarse points first and then forms the prolongation matrix such that each fine point relies on several coarse points. The other one is the aggregation-based method that only groups the strongly influenced points together. The aggregation-based method forms the disjoint coarse aggregations groups, so the fine points in the same group only rely on one coarse point. It gives one nonzero per row in the prolongation matrix, which is also called unsmoothed prolongation. Smoothed prolongation is obtained by applying matrix M (e.g., $I - \omega A$) to smooth original prolongation matrix P_0, i.e., $P = MP_0$. Figure 2.16 and Figure 2.17 briefly visualize the difference of coarse points and prolongation generations between classical methods and (unsmoothed) aggregation-based methods. In coarse points generations, classical methods select a subset of fine points as coarse points, but aggregation-based methods group the points together as coarse points in Figure 2.16. In generating prolongation step Figure 2.17, classical methods compute the values of fine points from more than one coarse point, but unsmoothed aggregation-based methods compute the values of fine points directly from the coarse point which they belong to.

2.5.2.1 Parallel Modified Independent Set (PMIS)

Hypre [FY02] introduced the parallel modified independent set algorithm (PMIS) [DSYH06] which is slightly different from Luby's parallel maximum independent set algorithm [Lub85]. The PMIS is one of the classical-based coarsening methods. PMIS is based on enforcing **H-2** and has a less stringent requirement **H-1'**[KS01]: Each

F-point needs to strongly depend on at least one C-point. Algorithm 4 gives the procedure to generate C and F points sets with PMIS. After we generate the C and

Algorithm 4 parallel modified independent set algorithm in [DSYH06]

```
1: procedure PMIS($A$)
2:     Generate weight $w(i) = \#S_i^T + rand([0,1])$
3:     Set F-point set F = $i \in V | \#S_i^T = 0$
4:     Set C-point set C = $\emptyset$
5:     Remove F-points in V, V' = V \ F and G' = g(V', S)
6:     while V' != $\emptyset$ do
7:         Choose the independent set I of V' in G', whose weight is larger than
   their neighbors. I = $i \in I | w(i) > w(j)$ with $i, j \in E'$
8:         Add I into C-points, C = C ∪ I
9:         Make all elements of V' \ I that are strongly influenced by a new C-point,
   F = F ∪ $j \in V' \setminus I | \exists i \in I : i \in S_j$
10:        Remove all new C-points and F-points from V', V' = V' C ∪ F and G' =
   g(V', S)
11:    end while
12: end procedure
```

F points sets, we can have three sets for each element i - C_i is the subset of the C-points used in prolongation for element i. They are usually the nearest neighbors of element i. D_i^s contains the points which strongly influence i, but are not in C_i and D_i^w of points connected to but not strongly influencing it. The following formula can construct the prolongation(interpolation) matrix,

$$w_{i,j} = \frac{-1}{a_{i,i} + \sum_{k \in D_i^w} a_{i,k}} \left(a_{i,j} + \sum_{k \in D_i^s} \frac{a_{i,k} a_{k,j}}{\sum_{m \in C_i} a_{k,m}} \right)$$

However, PMIS may encounter two strongly connected F-points that may not have a common C-point, which means the $\sum_{m \in C_i} a_{k,m}$ can be zero. Thus, the PMIS proposed another formula with an additional set: F_i is the set of strongly connected neighbors of i, which are F-points and do not share the common C-point,

$$w_{i,j} = \frac{-1}{a_{i,i} + \sum_{k \in D_i^w \cup F} a_{i,k}} \left(a_{i,j} + \sum_{k \in D_i^s \setminus F_i} \frac{a_{i,k} \hat{a}_{k,j}}{\sum_{m \in C_i} \hat{a}_{k,m}} \right)$$

, where $\hat{a}_{i,j} = 0$ if $sign(a_{i,j}) = sign(a_{i,i})$, otherwise $a_{i,j}$.

2.5.2.2 Parallel Graph Match (PGM)

PGM [Nau+15] is one of the aggregation-based methods. The algorithm tries to group the two nodes, which are each other's strongest neighbors, together (with aggregation size 2). If

$$|A_{i,j}| \geq |A_{i,k}|, \ \forall k \neq i,$$

the point j is the **strongest neighbor** of the point i. PGM algorithm in Algorithm 5 is the one we choose to evaluate the AMG implementation and mixed precision benefit, so we also detail the algorithm with more implementation detail in Algorithm 17.

Algorithm 5 Parallel Graph Match Algorithm in [Nau+15]

```
1: procedure PGM($A$)
2:     W is the set of vertex of A
3:     while W != $\{\phi\}$ do
4:         for $v \in W$ do
5:             Get the v's strongest neighbor w = sn(v)
6:         end for
7:         for $v \in W$ do
8:             if sn(w) = v then
9:                 Form aggregate i (set P matrix elements $P_{vi} = P_{wi} = 1$)
10:                Remove these vertex $W = W \setminus \{v, w\}$
11:            end if
12:        end for
13:    end while
14: end procedure
```

PGM has two phases: First, each node extends its hand to its strongest neighbor. Then, if they extend their hands to each other, they form the aggregation group. It uses unsmoothed aggregation, P matrix aggregates the nodes without considering the values. The P matrix is defined such that there is only one nonzero per row in P matrix.

2.6 Solvers and Smoothers

Building upon the previously presented dense and sparse matrix formats in Section 2.4, solvers solve a system of equations represented as a matrix. We introduce common solvers and smoothers used with multigrid such as the Conjugate Gradient method (CG), weighted Iterative Refinement (IR), Chebyshev Iteration, Jacobi, and ℓ_1-Jacobi. Besides these solvers or preconditioners, [Saa03] and [Bar+94] introduce several iterative solvers for sparse matrices additionally.

Solvers often need a preconditioner to help the convergence. Suppose we would like to solve x from a nonsingular linear system

$$Ax = b$$

For any nonsingular matrix M, the system

$$M^{-1}Ax = M^{-1}b$$

has the same solution. If we use an iterative solver to solve $M^{-1}Ax = M^{-1}b$, the convergence will depend on the properties of $M^{-1}A$ not A. If we choose the preconditioner M well, solving $M^{-1}Ax = M^{-1}b$ may be more rapid than $Ax = b$. To ensure it is useful in practice, we should be able to compute $M^{-1}A$ quickly. Usually, we do not form the M^{-1} explicitly but compute the solution y of the preconditioner system:

$$My = c$$

Conjugate Gradient method (CG) [Saa03] in Algorithm 6 is an efficient Krylov solver for a symmetric positive definite (SPD) system. The symmetric postive definite matrix A satisfies symmetric $A = A^T$ and positive definite $x^T Ax > 0,\ \forall x \in \mathbb{R} \setminus \{0\}$. It expands the Krylov subspace in each iteration. With the SPD property, CG only needs to search the minimization on the new Krylov vector because the new Krylov vector is orthogonal to the previous space. Thus, we do not need to keep the full Krylov space to solve the system.

Algorithm 6 Conjugate Gradient (CG)

```
 1: procedure CG($A$, $x$, $b$, $M$)
 2:     Compute $r = b - Ax$
 3:     Set $\rho_{prev} = 1$ and $p$ as zero vector with the same size of $x$
 4:     for i = 1, 2, ... do
 5:         Solve $z$ from $Mz = r$
 6:         Compute $\rho = r^T z$
 7:         Set $\beta = \frac{\rho}{\rho_{prev}}$
 8:         Compute $p = z + \beta p$
 9:         Compute $q = Ap$
10:         Set $\alpha = \frac{\rho}{p^T q}$
11:         Update the solution $x = x + \alpha p$
12:         Update the residual $r = r - \alpha q$
13:         Store $\rho_{prev} = \rho$
14:     end for
15: end procedure
```

Iterative Refinement (IR) [Bar+94] is a simple iterative scheme. It computes the residual and then obtains a solution for the residual by an approximated inner solver. We detail the algorithm step in Algorithm 7. With a weighted version, it can also be called modified Richardson Iteration.

Algorithm 7 weighted Iterative Refinement (IR)

```
1: procedure IR($A$, $x$, $b$, $\omega$, $M$)
2:     for i = 1, 2, ... do
3:         Compute the residual $r = b - Ax$
4:         Solve $z$ from $Mz = r$
5:         Update the solution $x = x + \omega z$
6:     end for
7: end procedure
```

Chebyshev Iteration [Bar+94] is a more advanced iterative refinement scheme. It is developed based on the first kind of Chebyshev polynomial. Unlike IR which relies on the weight parameter from users, Chebyshev Iteration requires knowledge of the eigenvalue region of the matrix. Algorithm 8 requires the maximum and minimum eigenvalue of the preconditioned matrix $M^{-1}A$.

Scalar Jacobi [Saa03] in Algorithm 9 is a kind of cheap preconditioner. It only needs to know the inversion of the diagonal part of the matrix.

ℓ_1-Jacobi is a ℓ_1 variant version of scalar Jacobi in Algorithm 10. The paper [Bak+11] introduces ℓ_1 variant that makes smoother use the matrix by adding the value of the

Algorithm 8 Chebyshev Iteration

```
1: procedure Chebyshev($A$, $x$, $b$, $\lambda_{max}$, $\lambda_{min}$, $M$)
2:     Set the center of eigenspace $c = \frac{\lambda_{max} - \lambda_{min}}{2}$
3:     Set the radius of eigenspace $r = \frac{\lambda_{max} + \lambda_{min}}{2}$
4:     Compute the residual $r = b - Ax$
5:     for i = 1, 2, ... do
6:         Solve z from $Mz = r$
7:         if i == 1 then
8:             Set $p = z$
9:             Set $\alpha = \frac{2}{d}$
10:        else
11:            Set $\beta = (\frac{c\alpha}{2})^2$
12:            Set $\alpha = \frac{1}{d - \beta}$
13:            Compute $p = z + \beta p$
14:        end if
15:        Update the solution $x = x + \alpha p$
16:        Update the residual $r = r - \alpha Ap (= b - Ax)$
17:    end for
18: end procedure
```

Algorithm 9 (scalar) Jacobi

```
1: procedure Jacobi($A$)
2:     Extract diagonal entries from $A$ to $D$
3:     Return the solver for $D^{-1}$
4: end procedure
```

off-diagonal block to the diagonal value to get better convergence. With the given block structure for smoother, we can have two sets for each row i. Diagonal block set $\Omega^{(i)}$ contains the column index of the same block as row i. Off-diagonal block set Ω_o^i contains the remaining column index out of the block of row i. Then, the ℓ_1 variant uses $D_{ii}^{\ell_1} = \sum_{j \in \Omega_o^{(i)}} |A_{ij}|$ for iteration update. In the scalar Jacobi version, each row i forms one block, so we get $\Omega^{(i)} = \{i\}$ and $\Omega_o^i = \{j | \forall j \neq i\}$. Thus, the ℓ_1-Jacobi uses $diag(A_{ii} + \sum_{j \neq i} |A_{ij}|)$ not $diag(A_{ii})$ in Algorithm 10

Algorithm 10 ℓ_1-Jacobi

```
1: procedure $\ell_1$-JACOBI($A$)
2:     Extract diagonal entries from $A$ to $D$
3:     Add the absolute value of off-diagonal to the diagonal matrix: $D_{ii} = D_{ii} + \sum_{j \neq i} |A_{ij}|$
4:     Return the solver for $D^{-1}$
5: end procedure
```

We have used the following type of smoother in Chapter 5 and Chapter 6:

- weighted Jacobi smoother: IR in Algorithm 7 with the Jacobi in Algorithm 9 as the inner solver(M)
- ℓ_1-Jacobi-chebyshev smoother: Chebyshev Iteration in Algorithm 8 with the ℓ_1-Jacobi in Algorithm 10 as the inner solver(M)

If the matrix is the symmetric positive definite (SPD), ℓ_1-Jacobi can ensure the eigenvalue bounded by 1 with the following formula.

$$\|M^{-1}A\|_\infty = max_i \sum_j |M_{ii}^{-1} A_{ij}| = max_i \sum_j \frac{|A_{ij}|}{\sum_j |A_{ij}|} = 1$$

, where M is the ℓ_1-Jacobi of A, so it is $diag\{A_{ii} + \sum_{j \neq i} |A_{ij}|\} = diag\{\sum_j |A_{ij}|\}$ when the matrix A is SPD matrix. Thus, if A is SPD, the eigenvalues of $M^{-1}A$ are in (0, 1]. With the property ensuring the bound of eigenvalue, we do not need to estimate the eigenvalue for the problem when using ℓ_1-Jacobi-Chebyshev smoother on SPD matrices. The paper [EHANN22] also uses this kind of smoother for their multigrid setup.

2.7 Hardware Used in This Dissertation

In this dissertation, we run the experiments on the following GPUs found in the largest HPC systems - AMD MI250X (1GCD), Intel Max1550 (1tile), and NVIDIA H100(PCIE). AMD MI250X contains two graphics compute dies (GCDs), but they are considered separately from the system's point of view. To use them together, we need to go through the MPI layer, so we consider 1GCD as the unit of MI250X. Intel Max1550 contains two tiles like AMD MI250X. Intel Max1550 provides the implicit scaling mode to use the two tiles as one GPU without an MPI layer. However, The implicit scaling splits the work and memory equally into two tiles, so it does not work well on the imbalance work, which is common with sparse matrix. The relatively

GPU	Peak Perf. (DP)	Peak Perf. (SP)	Mem. size	Bandwidth	Type
GPUs used in the experiments of this dissertation					
`AMD MI250X (1 GCD)`	24 TFLOP/s	24 TFLOP/s	64 GB	1.6 TB/s	Server
`NVIDIA H100 (PCIE)`	26 TFLOP/s	51 TFLOP/s	80 GB	2.0 TB/s	Server
`Intel Max1550 (1 Tile)`	22.8 TFLOP/s	22.8 TFLOP/s	64 GB	1.6 TB/s	Server
GPUs we use in papers we refer to					
`AMD MI100`	11.54 TFLOP/s	23.1 TFLOP/s	32 GB	1.2 TB/s	Server
`AMD RadeonVII`	13.44 TFLOP/s	3.36 TFLOP/s	16 GB	1 TB/s	Consumer
`NVIDIA V100 (SXM2)`	7.8 TFLOP/s	15.7 TFLOP/s	16 GB	0.9 TB/s	Server
`Intel UHD P630`	0.12 TFLOP/s	0.46 TFLOP/s	RAM	0.042 TB/s	Integrated

Table 2.3: GPU characteristics. In addition to the GPUs we use in this dissertation, we list also the GPUs we use in papers we refer to.

slow communication between the two tiles usually results in low performance in the sparse routine. Thus, we only focus on one tile as the base unit of Intel Max1550. We collect the GPU characteristics in Table 2.3.

We use the compiler and vendor libraries provided by the maintainers on these supercomputers as the following:

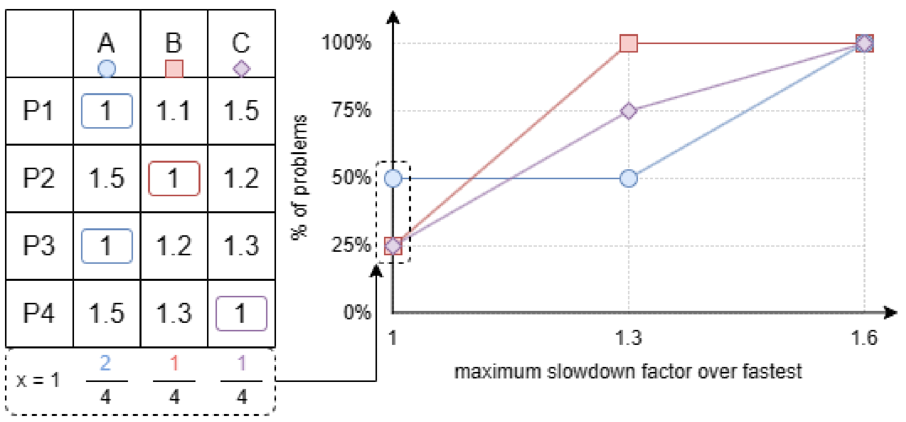
- CUDA environment: We use CUDA 12.0 as the device-side vendor compiler and GCC 8.5 as the host compiler to compile the application on NVIDIA H100 (PCIE) from BwUniCluster 2.0, bwHPC, Germany.
- HIP environment: We use HIP 5.3 as the device-side vendor compiler and Cray clang 15.0 as the host compiler to compile the application on AMD MI250X (1 GCD) from Frontier, Oak Ridge National Laboratory, USA.
- SYCL environment: We use DPCPP 2023.1 as the device-side vendor compiler and host compiler to compile the application on Intel Max1550 (1 tile) from Sunspot, Argonne National Laboratory, USA.

2.8 Performance Profile

Because we run our experiments on a large set of matrices, we need a way to summarize and compare performance on one graphic. In this dissertation, we use the performance profile [HH05]. The performance profile uses an x-axis for the slowdown factor and a y-axis for the ratio of the problems that can be solved by a certain algorithm within the slowdown factor times of the best timing. The value y is decided by the algorithm and given x, that is,

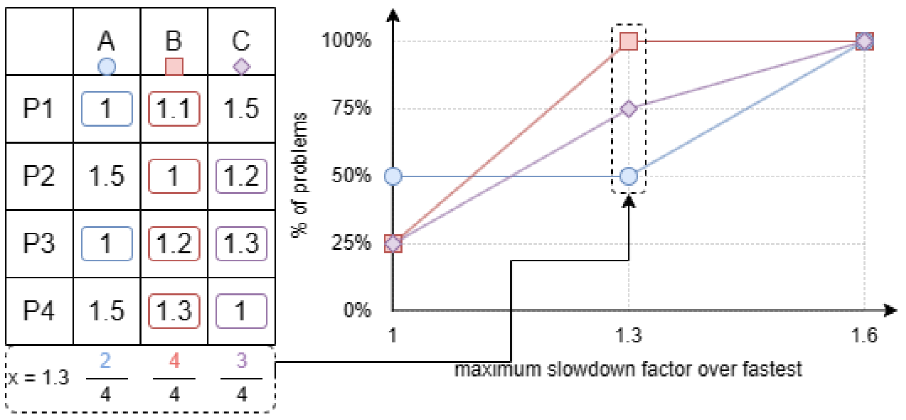
$$y(x, algorithm) = \frac{\sum_{\forall problems}\{time_{algorithm} < x \times time_{best}\}}{\#\{total\ problems\}}$$

We use the Figure 2.18 and Figure 2.19 as an example to describe the performance profiles. There are performance results from 3 algorithms (A, B, and C) on 4 problems. Without loss of generality, we normalize the time over the best time per problem. Considering the accepted slowdown factors 1, 1.3, and 1.6, we can have the performance profile as the right side of Figure 2.18 for the data as the left side of Figure 2.18. For x = 1, it indeed plots the ratio for the best performance among



	A	B	C
P1	1	1.1	1.5
P2	1.5	1	1.2
P3	1	1.2	1.3
P4	1.5	1.3	1
x = 1	$\frac{2}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Figure 2.18: The performance profile example with a detail view for x = 1.



	A	B	C
P1	1	1.1	1.5
P2	1.5	1	1.2
P3	1	1.2	1.3
P4	1.5	1.3	1
x = 1.3	$\frac{2}{4}$	$\frac{4}{4}$	$\frac{3}{4}$

Figure 2.19: The performance profile example with a detailed view for x = 1.3.

the algorithm, so we get 50%, 25%, and 25% for A, B, and C, respectively, as shown in the highlighted part of Figure 2.18. Figure 2.19 shows the selected cases for each algorithm when the maximum slowdown factor is 1.3. We pick all cases whose time is less than 1.3 for x = 1.3 because the best time for all problems is 1 by normalization. We get 50%, 100%, and 75% for A, B, and C, respectively. Doing this calculation for all values of the slowdown factor gives us the performance profile in the end.

The performance profile does not only show the best algorithm (for $x = 0$) but also shows the best choice when allowing some slowdown or wanting a more general solution. From the example, algorithm A is the best one in terms of absolute performance. However, algorithm B is a more general solution. If we choose algorithm B, we will only get a 1.3x slowdown at most for all problems. However, choosing algorithm A may give a 1.5 slowdown in some cases. The performance profile gives us a different view of the algorithm selection with different slowdown factor allowances under the problem set. Besides the typical performance against nonzero plots, we also use the performance profile when analyzing the performance of the distinct algorithms for the complete set of matrices available in the Suite Sparse Matrix Collection[DH11] in Chapter 4.

3. Ginkgo

Designing multigrid is complex with the hierarchy and many options for configuration. We need a software library with a flexible interface to design the flexible multigrid, which should be available on AMD, Intel, and NVIDIA GPUs. Similarly, the library must support different precisions such that we can add the mixed-precision idea into AMG. We also identified that no AMG implementation is portable to the GPU hardware from supercomputers, which we would like to address in this dissertation. We chose the ginkgo, which has sustainability and portability in the design and aims for high-performance computing.

Efficient linear algebra libraries on GPU are an essential and important part of many applications with high-performance computing interests. Moreover, software sustainability, lifecycle, and correctness are considered more seriously nowadays. There are several aspects and techniques, such as continuous integration/delivery (CI/CD), unit tests, and portability, to improve libraries with these concerns. At the same time, many projects have started to use object-oriented software design like C++ to adopt more advanced functionalities and safer memory management.

GINKGO is one of the libraries that follow these concerns and provides its own design to provide a more stable, usable, high-performance sparse linear algebra library for multicore and manycore architectures. GINKGO uses the platform's native language with architecture-specific kernel optimization to build a high-performance library. It directly uses CUDA for NVIDIA GPUs, HIP for AMD GPUs, SYCL (oneAPI) for Intel GPUs, and OpenMP for general-purpose multicore processors. GINKGO strictly requires two reviewers for pull requests, follows the semantic version, contains the unit tests, and sets up a long list of jobs for the CI/CD pipeline to ensure the software maintains production quality. GINKGO is an open-source effort licensed under the BSD 3-clause license[1]. GINKGO follows the community guidelines like xSDK project [Bar+17], E4S [E4s] and the Better Scientific Software (BSSw [Bet]) initiative

[1] https://opensource.org/licenses/BSD-3-Clause

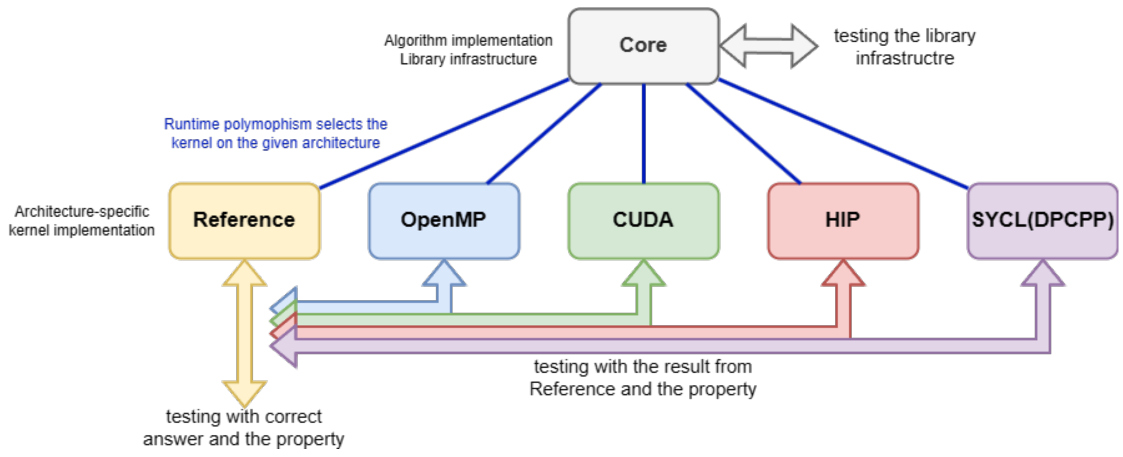

Algorithm implementation
Library infrastructure
Core
testing the library infrastructre
Runtime polymophism selects the kernel on the given architecture
Architecture-specific kernel implementation
Reference
OpenMP
CUDA
HIP
SYCL(DPCPP)
testing with the result from Reference and the property
testing with correct answer and the property

Figure 3.1: Ginkgo library design separating the core containing the algorithms from architecture-specific backends.

for the sustainability. Several libraries and applications are already using Ginkgo or prepare to use Ginkgo such as SUNDIALS[Gar+22; Hin+05], OpenFOAM[Ope; Wel+98], OpenCARP[ope+23; Pla+21], NekRS[Fis+22], MFEM[And+21; Mfe], deal.ii[Arn+21] and hiop[PCW18].

In this chapter, we describe the design principles from Ginkgo library. Ginkgo designed the backend model for portability and performance, and linear operator abstraction for composable components. We extended the backend model from NVIDIA GPU to support AMD and Intel GPU by addressing the challenges from hardware and ecosystem differences in Section 2.1. The linear operator abstracts the components' interface such that we can reuse the components in several places easily. Under this linear operator flexible interface, we provide high-performance SpMV but extend the mixed-precision support in Chapter 4. We follow the linear operator flexible interface to design the flexible AMG in Chapter 5 and extend to mixed-precision AMG in Chapter 6.

3.1 Overview of Ginkgo's design

3.1.1 Executor and Backend

To use platform-specific ecosystems for different devices, we split the library into the core part for the generic, hardware-independent algorithm implementations and the backend part for the hardware-specific kernel implementations in Figure 3.1. Note that the core part does not know which device will be used in compile time because the backend selection is done by the user at runtime. We use runtime polymorphism to select the actual function on specific devices to connect the core and backend implementation. The backend libraries are not embedded in the core library such that we can use different compilers to compile each backend and then link them together in the end. We will introduce the testing structure shown in Figure 3.1 in Section 3.3

The Executor class provides the information about which backend is selected. We have the following Executor support:

```
// core
// It will call different kernel based on runtime Executor type
this->get_executor()->run(make_kernel(...));

// reference
void kernel(std::shared_ptr<const ReferenceExecutor> exec, ...) {
    // implementation for reference backend
}

// cuda
void kernel(std::shared_ptr<const CudaExecutor> exec, ...) {
    // implementation for cuda backend
    // cuda_kernel<<<...>>>(...);
}

// also for the other backend
```

Listing 3.1: Executor component implementation concept

```
auto exec = gko::ReferenceExecutor::create();
auto dev_exec = gko::CudaExecutor::create(0, OmpExecutor::create());
// auto dev_exec = HipExecutor::create(0, OmpExecutor::create());

auto host_matrix = gko::matrix::Csr<>::create(exec);
auto dev_matrix = gko::matrix::Csr<>::create(dev_exec);
```

Listing 3.2: The user can simply set up an executor to use and the application works smoothly if changing it to a different executor.

- ReferenceExecutor for sequential CPU execution
- OmpExecutor for multicore processor
- CudaExecutor for NVIDIA GPUs
- HipExecutor for AMD GPUs
- DpcppExecutor for Intel GPUs/CPUs and other SYCL hardware

Each executor contains its own memory allocation/deallocation/copies, and GINKGO implements the corresponding kernels for these executors as in Listing 3.1. When the core library wants to use a `kernel` function via GINKGO'S own selector `make_kernel`, each backend library needs to provide the implementation for the `kernel` function. In the GINKGO library configuration, users can specify backends or rely on auto-detection, in which GINKGO will compile the backends based on the available compilers on the system.

From the users' perspective, they only need to set up the executor for their devices and use the executor for the operations. For example, in Listing 3.2, users can set up a host executor and device executor. The functionalities' usage is the same, and they will return the correct object based on the given executor. Giving ReferenceExecutor results in an object on the memory of a ReferenceExecutor (i.e., CPU memory), but giving CudaExecutor results in an object on the memory of a CudaExecutor (i.e., NVIDIA GPU memory). Suppose users want to test the same codes on different platforms. In that case, they only need to change the CudaExecutor to the HipExecutor without changing anything else to use AMD GPUs.

3.1.2 LinOp and LinOpFactory

LinOp is one of the most important classes in Ginkgo, representing linear operators' usage. The components in solving linear systems are the matrix format, preconditioner, factorization, iterative solver, and Krylov solver. The LinOp can unify the application programming interface (API) for these linear operators. The following list denotes how we define the linear operator:

- **Matrix Format**: The matrix A gives the linear operator $L_A : x \to Ax$
- **Solver**: The solver solves the linear system $Ax = b$, which gives the linear operator $S_A : b \to A^{-1}b$. S_A is based on the fixed system matrix A.
- **Preconditioner**: The preconditioner uses the application $y = M^{-1}x$, which is $P_M : x \to M^{-1}x$. It is similar to the solver, but preconditioners sometimes only take part of matrix A or modify the matrix A such that we use M, not A directly here.
- **Factorization**: the factorization generates a composition of linear operators, so it is still linear operators.

We need to emphasize the above statements are only based on theory. In practice, the linear operator we introduced does not strictly fit the definition $L(\alpha x + \beta y) = \alpha L(x) + \beta L(y)$, where α, β are scalars and x, y are vectors. Several effects affect it, such as the rounding error from finite precision arithmetic and the inexact application (when doing only a few iterations of an iterative solver).

LinOp is the abstraction of linear operators in Ginkgo and provides the uniform interface `apply`. Each LinOp contains two kinds of apply function: apply: `L->apply(x, y)` is y = L(x) and advanced apply: `L->apply(`α`, x, `β`, y)` is y = αL(x) + βy, where α, β are scalars and x, y are vectors. With LinOp's help, we can use different components of Ginkgo uniformly and flexibly to solve a problem. For example, a conjugate gradient (CG) solver only needs to rely on the subcomponent's apply function so that we can give any matrix format shown in Section 2.4 or we can define a matrix-free linear operator without storing the matrix explicitly.

The LinOpFactory type is a way in Ginkgo to configure and generate the LinOp tailored for these parameters. C++ does not natively provide the optional key-value input with any order and type. Thus, we have the LinOpFactory to handle the options because the parameters in solvers/preconditioners should be optional and contain default values. For example, we can generate a conjugate gradient (CG) solver with/without a preconditioner or different stopping criteria. The factory uses a specific prefix "`with_`" for the key. In Listing 3.3, `build()` creates the parameters set class. Users can set the preconditioner by using "`with_precontioner`", and criteria by using "`with_criteria`". Those parameters are optional, so they contain default values if they are not set. For example, the current settings of Listing 3.3 give preconditioned CG, but it becomes a plain CG if we omit line 2 `with_preconditioner(...)`. The criteria setting allows the users to specify one or many conditions. We will introduce the detail of the criterion later in Section 3.1.3. After setting everything up, "`on(exec)`" generates the LinOpFactory on the "exec" Executor, which indicates the usage location. We can use the LinOpFactory to

```
auto cg_factory = gko::solver::Cg<>::build()
    .with_preconditioner(linopfactory)
    .with_criteria(stop_1, stop_2)
    .on(exec);
auto cg = cg_factory->generate(A);
// solve Ax = b by CG with initial guess x
cg->apply(b, x)
```

Listing 3.3: LinOpFactory provides the optional key

```
auto stop_factory = gko::stop::ResidualNorm()::build()
    // default is 1e-15
    .with_reduction_factor(1e-14)
    // default is based on the right hand side
    .with_baseline(mode::absolute)
    .on(exec);
```

Listing 3.4: Set up absolute residual norm stopping criterion with 1e-14.

generate the LinOp by calling `generate(linop)` at the end of Listing 3.3, which gives the configured solver S_A. The LinOpFactory can generate several LinOp with different LinOp inputs thanks to the generate call. The generated LinOp always relies on the child LinOp to provide its apply function. Because CG requires an initial guess, we use the output vector as the initial guess in the beginning and then update the solution in the output vector.

3.1.3 Stopping Criterion

Another important component for solvers is controlling when to stop the solver. The stopping criterion can be based on a maximum number of iterations, a time limit, or a residual criterion. We have the implicit residual criterion for some solvers like CG, which provides the internal residual in the computation, and the explicit residual criterion for all kinds of solvers, which is an actual residual from the current solution. Listing 3.4 shows the setting for a stopping criterion requiring 1e-14 for the absolute residual norm. Ginkgo uses the same idea as LinOpFactory to design the stopping criterion interface for the optional parameters settings. If users do not provide any parameters, the default will be a relative residual norm reduction of 1e-15 of the right-hand side norm. Ginkgo allows one or many criteria for a solver.

3.2 Portability

The lifetime of software is usually longer than the lifetime of hardware. Supporting the libraries on different vendors' devices improves the usability and lifecycle of software. As Ginkgo is the essential core component of numerical applications, the portability onto different devices helps the users explore the new device's possibilities and not be limited by a certain vendor. However, portability is a big challenge for the performance-aimed library for the following reasons:

- **Hardware**: Different vendors have their own target user group and focus such that they have different strategies for designing their hardware. The different designs usually mean different techniques to use the device efficiently.

- **Ecosystem**: Vendors develop their own ecosystems to use the device. There may be some common ecosystems for multiple vendors, but the new techniques are only in their ecosystems first. For example, HIP can also compile for NVIDIA GPUs, but it can not use dynamic parallelism or 16-bit atomic operations. SYCL can also support AMD/NVIDIA or other vendor devices as a portability layer, but with different limitations, like subwarps not being officially supported in SYCL.

- **Software stack/library support**: Vendors' support may differ, such that the libraries must implement the missing components. CUDA uses cuSPARSE as the vendor sparse library for NVIDIA GPUs, HIP uses hipSPARSE(rocSPARSE) for AMD GPUs, and SYCL uses oneMKL for Intel GPUs. For example, rocSPARSE/hipSPARSE and oneMKL do not support the half-precision SpMV, which is discussed in Section 4.2

Ginkgo focuses on the three main GPU vendors in high-performance computing - AMD, Intel, and NVIDIA. Ginkgo was originally designed for NVIDIA devices, and then it brought HIP support for AMD devices from version 1.2.0 and SYCL support for Intel devices from version 1.4.0. The Ginkgo's developers are familiar with the CUDA ecosystem. Thus, our portability effort considers not only the supported functionalities but also the code readability for CUDA developers. AMD and Intel provide the porting tools "hipify" and "dpct"(Intel DPC++ Compatibility Tool) to transfer CUDA code to HIP and SYCL codes. We also draw the evolution of Ginkgo with more information in Figure 3.6. "dpcpp" is a compiler Intel provided for SYCL. We use the term "dpcpp" for the SYCL backend in Ginkgo.[2]

3.2.1 Adding a New Backend

We develop Ginkgo with portability as the central design principle. After the first major release with support only for NVIDIA, we extended Ginkgo for different platforms like Intel and AMD GPUs. The different platform adoption does not change the public interface, which only requires new minor revisions according to the semantic versioning introduced in Section 3.3. We designed and used the following general workflow to add a new backend:

1. Add a new executor that defines the memory interaction with the other executors as well as how to launch kernels on the device.

2. Setup compilation rules for the new ecosystem with dummy kernels.

3. Implement core components like the cooperative groups or atomic operation wrappers.

4. Implement/Port kernels one by one for the new backend.

5. Optimize the code structure and performance.

[2]As Intel deprecated the dpcpp compiler, we consider renaming this backend name to SYCL. In this dissertation, we use SYCL to describe the backend for Intel GPUs.

The first two steps make GINKGO able to run on the new ecosystem with the new devices. The latter two steps ensure that the ecosystem can fit the GINKGO's requirements, and expand the functionality support by porting or implementing the function one after the other. Furthermore, we also optimize the kernel performance to utilize devices fully. We repeat the last two steps to refine and add the codes iteratively to improve GINKGO.

3.2.2 CUDA to HIP

One of the most essential differences between NVIDIA and AMD GPUs is the warp (or wavefront) size. The warp size is 32 for NVIDIA devices, but the wavefront size is 64 for AMD server devices. Warp and wavefront allow the memory exchange at the register level, which is a lightweight memory operation compared to the shared or global memory. GINKGO develops several high-performance kernels based on this important size as a core feature. As we describe the design in Section 3.1.1, HIP codes and CUDA codes are compiled separately. We specify a compile-time constant in the headers for these different sizes and carefully deal with the difference between block size and memory size. For example, although the wavefront size is 64, the block size only allows a maximum of 1024 in AMD GPUs. It introduces the illegal block size (2048) if we use the whole wavefront for the kernels requiring squared wavefront size like dense matrix transpose. In this case, we need to restrict the kernel to still use size 32 x 32 not 64 x 64 on AMD GPUs.

HIP did not provide the cooperative group interface[3] when we ported the GINKGO CUDA backend to HIP backend. However, the cooperative group is a good abstraction to avoid painful suffix or mask calculations, which CUDA 9 introduced, and we use it a lot for warp memory operations. The cooperative group allows us to focus on the operation of the current group, not the entire warp when we write small sub-warp operations. GINKGO cooperative group needs to support register swapping (such as `shuffle, shuffle_up, shuffle_down, shuffle_xor`) and the voting functions (such as `ballot, any, all`) via registers in a warp. Different synchronization functions from different levels of groups are also required. Furthermore, GINKGO adds the identifiers for the grid, block, and warp levels to help the usage of the C++ pattern "substitution failure is not an error (SFINAE[Sfi])" in kernel design. Thus, we provide our own cooperative group implementation by wrapping the low-level HIP API call with an interface similar to CUDA. We also show that our cooperative group does not introduce overhead in Figure 3.2 from [TCA20].

Another portability issue is that `__launch_bound__` has a different meaning between CUDA and HIP. `__launch_bound__` can give information as a hint to the compiler for optimizing the kernel under the given limit. CUDA uses `__launch_bound__(max_threads_per_block, min_blocks_per_multiprocessor)`, but HIP uses `__launch_bound__(max_threads_per_block, min_warps_per_execution_unit)`. HIP also provides the corresponding formula for the second input: min_warps_per_execution_unit = (min_blocks_per_multiprocessor * max_threads_per_block) / 32. Fortunately, GINKGO mainly uses the first parameter only, so we rarely need to deal with this difference.

HIP provides math libraries with the same interface as CUDA for almost all functionalities but changes the name from CUDA to HIP. For example, cuSPARSE for sparse

[3]HIP starts to support some of the cooperative group from version 4.5.0

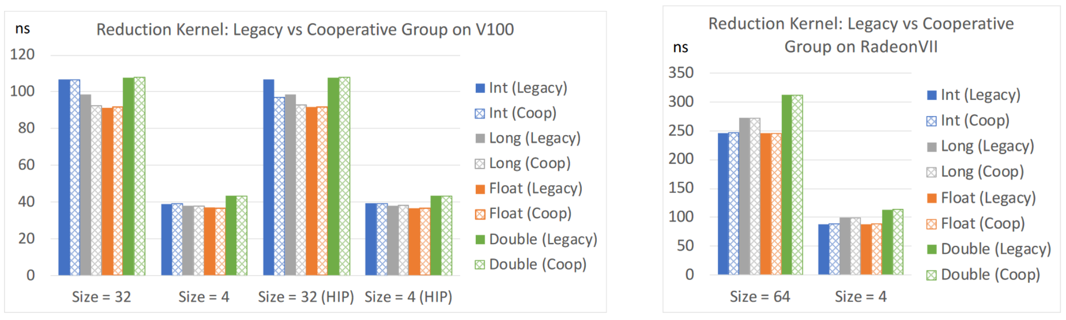


Figure 3.2: Ginkgo's cooperative groups vs. legacy functions for different data types on V100 (left) and RadeonVII (right) from [TCA20]

functionality becomes hipSPARSE. With Ginkgo's cooperative groups, the device kernels look similar up to some configuration changes. We can simply copy those device codes to the HIP side with the changed parameters, but duplicating the almost identical codes is not good for maintainability. Thus, we extract the device kernels to the shared folder "cuda_hip" between CUDA and HIP. The "cuda_hip" folder only contains the codes without any parameter settings. Thus, we use "cuda_hip" to represent the shared components between CUDA and HIP in Figure 3.6. Before including these shared codes, we define the parameters set for optimization or devices. By doing so, we keep the flexibility for optimization and reduce the duplication. We have a more detailed discussion and the performance discussion about using HIP on NVIDIA GPUs in [TCA20; Tsa+21]. All in all, only using specific non-portable intrinsics or relying on different hardware features requires separate kernels between CUDA and HIP.

3.2.3 CUDA to SYCL(oneAPI)

Unlike HIP, SYCL is very different from CUDA. From Table 2.1, we see that SYCL does not share the same technical terms as CUDA or HIP. SYCL also introduces several differences in kernel submission, index of work-items, shared memory usage, and the sub-group(warp) concept. We can not use the same shared code strategy as HIP for SYCL. We make the ported code close to CUDA-style and support most functionalities. We also use the portability tool - "dpct" provided by Intel. "dpct" was a closed-source tool when we ported to SYCL. We designed several approaches to port our code quickly and fit our requirements by constraining the tool's behavior. Since then, Intel deprecated the tool and replaced it with SYCLomatic [Syc].

Difficulty with the porting tool: Unlike CUDA and HIP, which provide index information like `threadIdx.x` without any arguments in the function, SYCL must pass the extra argument `nd_item` to the function and access the index information from that object. Sometimes, the access of index information may be in a deep code, so we need to pass `nd_item` from the top level to that function. It is a big challenge to handle this manually. "dpct" helps us to add the 'nd_item' for the indexing information, also deals with the shared memory pointer (see Section 2.1 in Chapter 2), and changes the name of the index if the function requires the information. However, "dpct" tries to access all local files to know the complete library to decide whether to add the nd_item or not. However, "dpct" will stop the porting process if it faces too many issues. It also stops even if the issues are in different files. As a big library, we prefer porting codes step by step such that we have small pull requests. Thus, we isolate the target code and treat the other headers as system headers to avoid these challenges.

Workaround to enable porting smoothly: However if we hide all functionalities as system headers, dpct will not help us to add the index information automatically. For those device kernels, we keep them as local headers if they do not introduce an issue to dpct. Suppose the device code introduces an issue to dpct. In that case, we design a method - a fake interface such that we can provide our own implementation for SYCL and dpct can still help us with index passing. Because dpct can not port these codes to SYCL, we need to prepare their SYCL version on our own first. We prepare fake interfaces that use the same interface without any actual implementation but add `auto x = threadIdx.x;` only if the kernels need the indexing information. We replace the function name and corresponding headers with these fake interfaces as local headers in preprocessing. When dpct ports the codes, it knows the functions require the index information and adds the nd_item for us. There is only one line `auto x = threadIdx.x;` in fake interfaces, so dpct converts them easily and successfully. In post-processing, we replace the fake interface with our own implementation. Thanks to the fake interface workaround, we successfully port all our CUDA kernels to SYCL and still rely on dpct's help.

The cooperative groups in SYCL: As Section 3.2.2 mentioned, GINKGO uses the cooperative group a lot and the feature is important for performance. We also provide a cooperative group wrapper for using the SYCL sub-group for register exchanges. We also use this workaround to make dpct port the CUDA code smoothly. Without this workaround, dpct always stops at the cooperative group and complains that it is not supported. We visualize the fake interface trick and the kernel porting process in Figure 3.3. The blocks of the top two rows show that actual porting only touches the files we provide, not the actual CUDA implementation which introduces failure to dpct. We provide the equivalent implementation for SYCL from CUDA shown on the right side. After porting, we replace these function calls with the proper function name so that it can use the working SYCL implementation at the bottom of Figure 3.3. Due to the `threadIdx.x` trick, dpct still adds `nd_item (item_ct1)` in the function call at the blue and yellow part.

Although we provide the cooperative group interface to exchange the register data in sub-groups, it can not exceed the vendor support. SYCL does not support different "sub-sub-group" communicating in a sub-group at the same time. We can not use the cooperative group in SYCL like the subwarp in CUDA. However, Intel GPU provides two sub-group sizes (16, 32) in newer GPU. We can use them as sub-sub-group sizes virtually if we do not need the communication out of this size within a larger sub-group. The single thread case with sub-sub-group size = 1 does not need any actual communication such that we can handle it without any sub-group functionalities. Thus, we can use (1, 16, 32) as virtual sub-sub-group size options on Intel GPUs.

SYCL does not support early return with synchronization: Another problem to be aware of is early return usage in SYCL. SYCL requires all work-items to arrive at any barrier, but CUDA only considers the living threads. We need to rewrite the conditions like Listing 3.5 to ensure the work-items do not return early. On the top of Listing 3.5, CUDA kernels only need to use one condition to terminate the

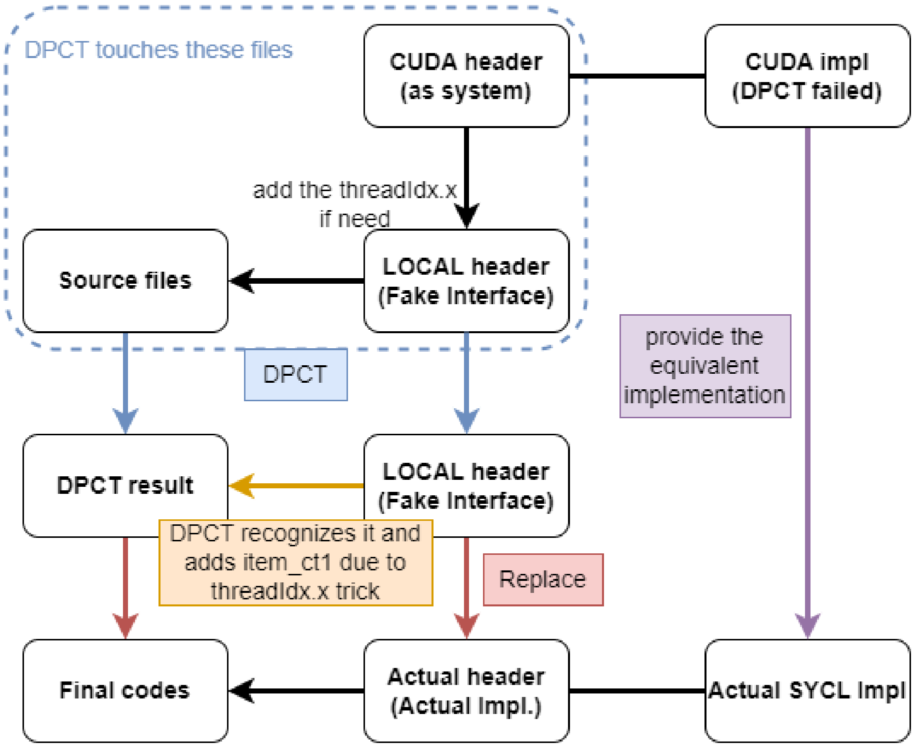

DPCT touches these files
CUDA header (as system)
CUDA impl (DPCT failed)
add the threadIdx.x if need
Source files
LOCAL header (Fake Interface)
DPCT
provide the equivalent implementation
DPCT result
LOCAL header (Fake Interface)
DPCT recognizes it and adds item_ct1 due to threadIdx.x trick
Replace
Final codes
Actual header (Actual Impl.)
Actual SYCL Impl

Figure 3.3: Using the fake interface to provide our SYCL implementation for porting.

unused threads. However, in the SYCL kernels on the bottom, we need to declare the variables in the top levels if more than one if-block needs the variables. We also need to use many if-blocks with corresponding barriers. Both of these problems make the code more complex. Intel recognized the early return issue and the lack of sub-sub-group, so they started to implement an extension to solve them.

SYCL kernel launch: As previously shown in Section 2.1, SYCL gives a different view on kernel submission and indexing. To provide a CUDA-style kernel submission interface, we implement our own kernel launch command and handle some simple conversions in an additional host function. Assuming the kernel requires dynamic and static shared memory, we use Listing 3.6 to describe the additional host kernel. SYCL has a dimension scheme that is a reversed version of CUDA/HIP in terms of the memory access pattern. The porting tool dpct always converts the `dim3(x, y, z)` to `nd_range(z, y, x)` to overcome this issue in line 9, 10 of Listing 3.6. We use a `dim3` interface for SYCL in the additional host function, and then it is converted to the proper range in the kernel launch command. The conversion also includes the preparation of `nd_range` in line 22. SYCL allocates shared memory in the kernel submission before launching kernels, which leads to a long code for kernel submission. We also put the shared memory allocation in the additional host functions. Although we need an additional host interface to provide CUDA-style interfaces, it gives the same view when using the kernel. It is easier to check the CUDA and SYCL code at the same time because we do not need to reverse the grid/block and the kernel submission has the same ordering of input arguments.

Figure 3.4 visualizes the original SYCL kernel call and its difference from CUDA and HIP in terms of shared memory allocation, parallel resource assignment, and kernel placement. With the additional layer of Figure 3.5, we provide the same interface for

```
// CUDA kernel
__global__ void func(...) {
    if (condition) return; // early return
    // some work
    __syncthreads();
    // some work
}

// SYCL kernel
void func(..., nd_item<3> item_ct1) {
    // Need to declare some variables if it is needed in both work block.
    if (!condition) {
      // some work
    }
    item_ct1.barrier(...);
    if (!condition) {
      // some work
    }
}
```

Listing 3.5: SYCL does not allow early return

the top kernel call among CUDA, HIP, and SYCL. The additional layer calculates the parallel resource assignment from CUDA dim3 to SYCL nd_range and avoids the shared memory allocation expanding the main code length. This effort is mainly for porting, and there is no restriction on the style in GINKGO. Thus, developers can still write the SYCL-style codes in GINKGO if they prefer.

Maintenance challenges: Because of the big difference between SYCL and CUDA, we can not provide a unified way to reduce the codebase. Instead, we develop a C++-lambda-based framework for simple component-wise kernels and reductions. Because component-wise kernels do not require any data exchange among the threads and do not need heavy optimizations, we decided to extract them to this unified framework to reduce the codebase size. The reduction abstraction only supports different operations between the elements and the final output operation, so we also put the optimized implementation into the unified framework. The unified kernels allow us to write the code once, and then we can generate the corresponding kernels for different backends thanks to CMake[Cma]. These are still distinct kernel implementations for different backends from the compilers' perspective, unlike other C++-lambda abstraction layers such as Kokkos[CETS14]. Thus, we have also represented the unified components for OpenMP, CUDA, HIP, and SYCL in Figure 3.6. We have also discussed more details about SYCL challenges in [TCA22; TCA23] and discussed CUDA/HIP/SYCL together in [CTA22]

```cpp
// CUDA launch kernel
cuda_kernel<<<gridsize, blocksize, dynamic_memory_size, stream>>>(...);

// SYCL launch kernel (originally)
queue->submit([&](sycl::handler& cgh) {
    sycl::accessor<....> dynamic_shared_memory(cgh);
    sycl::accessor<....> static_shared_memory(cgh);
    cgh.parallel_for(
        sycl::nd_range<3>(range<3>(1, 1, gridsize) * range<3>(1, 1, blocksize),
                          range<3>(1, 1, blocksize)),
        [=](sycl::nd_item<3> item_ct1) {
                sycl_kernel(..., item_ct1, dynamic_shared_memory,
                    static_shared_memory);
            });
});

// additional host kernel
void sycl_kernel(dim3 grid, dim3 block, size_t dynamic_memory_size, queue_t* queue,
     ...) {
    queue->submit([&](sycl::handler& cgh) {
        sycl::accessor<....> dynamic_shared_memory(cgh);
        sycl::accessor<....> static_shared_memory(cgh);
        cgh.parallel_for(
            sycl_nd_range(grid, block),
            [=](sycl::nd_item<3> item_ct1) {
                    sycl_kernel(..., item_ct1, dynamic_shared_memory,
                        static_shared_memory);
                });
    });
}

// gko's cuda-style launch kernel
sycl_kernel(gridsize, blocksize, dynamic_memory_size, queue, ...);
```

Listing 3.6: The proposed new style for kernel submission in GINKGO's SYCL backend

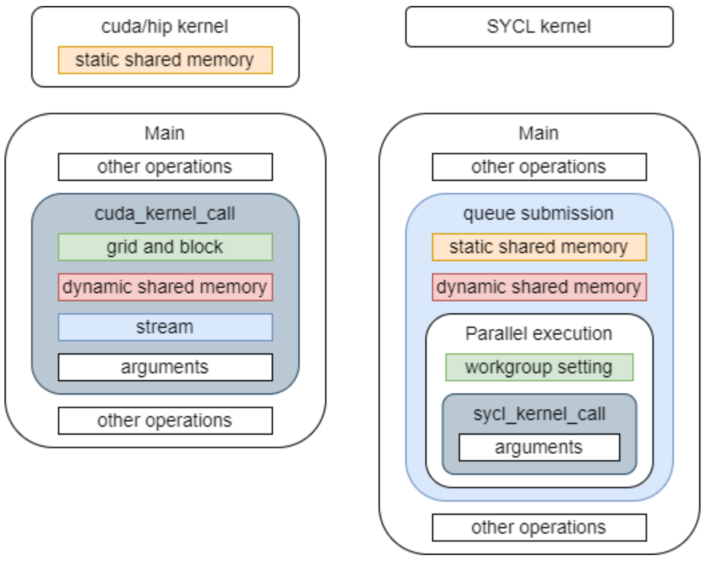


Figure 3.4: Different kernel calling styles between CUDA/HIP(on the left) and SYCL(on the right).

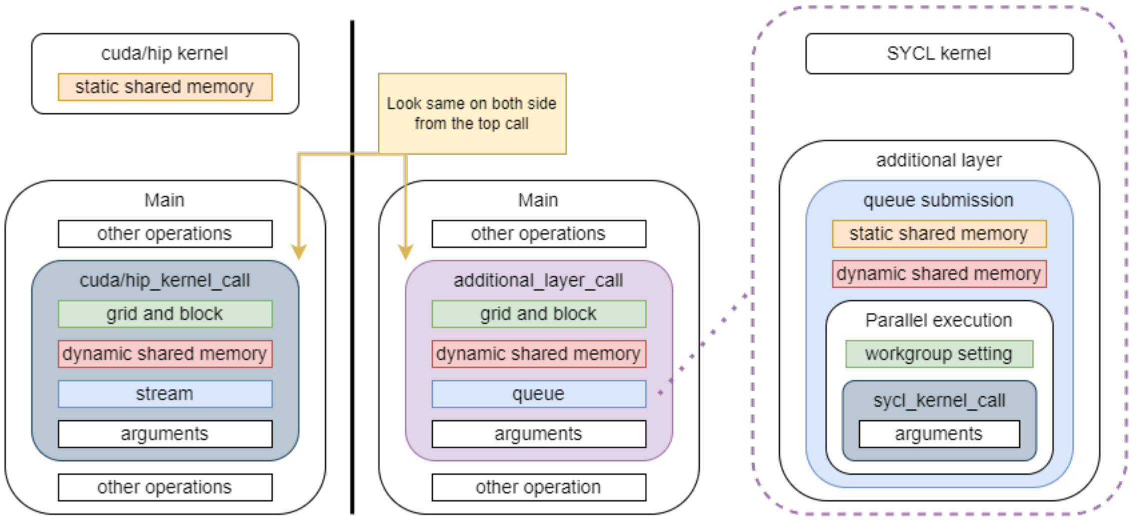


Figure 3.5: With an additional layer for SYCL, the top calling layer for kernels looks the same among Ginkgo's backends.

3.3 Sustainability

Software sustainability is also an important aspect of the Ginkgo library. Ginkgo follows the existing guidelines and policies from the xSDK project [Bar+17], E4S [E4s] and the Better Scientific Software (BSSw [Bet]) initiative. Also, Ginkgo matches the FAIR Principles for Research Software (FAIR4RS in [Hon+22]). FAIR defines the following four principles:

- **(F)indable**: Software and its associated metadata are easy for both humans and machines to find.
- **(A)ccessible**: Software and its metadata are retrievable via standardized protocols.
- **(I)nteroperable**: Software interoperates with other software by exchanging data and/or metadata, and/or through interaction via application programming interfaces (APIs), described through standards.
- **(R)eusable**: Software is both usable (can be executed) and reusable (can be understood, modified, built upon, or incorporated into other software).

For easy adoption, Ginkgo is open source on GitHub with the modified BSD license, which can also be used for commercial purposes. We also have the following requirements to ensure that we have a production-ready code quality.

- Unit testing: we use the googletest[Goo] to implement for unit tests and functional tests. It is also an important criterion in the peer review process. From Ginkgo's design in Figure 3.1, we use the reference backend for serial execution and the unit testing of this backend can check the correctness in small cases and the correct properties. Based on the correct reference implementation, we have the unit testing to compare the results between the parallel implementations (OpenMP, CUDA, HIP, SYCL) and the reference implementation under a certain precision threshold.
- CI/CD: We set a large set on GitHub action pipeline mainly for Windows and MacOS, and GitLab pipeline for Linux on customized runners with actual GPUs. The pull request must pass all tests before being merged. Other than unit testing, Ginkgo also contains software quality helpers like codecov [Cod] for testing coverage rate and the sanitizers to detect illegal memory access and thread issues.
- Documentation: We use Doxygen [Dox] to generate the documentation based on the comments above the corresponding codes, and enforce correct documentation of the public API.
- Semantic versioning: We follow the rule of semantic versioning, so we do not break the public interface when releasing minor versions unless it fixes fatal bugs. The semantic version gives three numbers for a given version in [Pre]: Major.Minor.Patch, and this combination must be unique. The numbers increase according to the following rules.

  - Major: on an incompatible API change.
  - Minor: when adding functionality in a backward compatible manner.
  - Patch: when making backward compatible bug fixes.

- Peer review: It requires the code to be understandable to others. The reviewers may catch some fatal bugs, recognize missing parts, or provide better optimization ideas. Also, the reviewers need to check the above concerns in the pull request. As [ACK19] mentioned, peer reviews are challenging to scientists due to the time pressure and academic credit system, but it is still useful for sustainability and quality. Ginkgo requires at least two peer reviews for every pull request.

With the GitHub repository and JOSS publication [Anz+20a], Ginkgo achieves the Findable and Accessible principles. Ginkgo reads the user data from matrix market format [BPR96] and JavaScript Object Notation (JSON [Jso]), which achieves the Interoperable principle as well as providing APIs which care for user data. Ginkgo's CI pipelines, spack and vcpkg packages, and several integrations such as MFEM, SUNDIALS, deal.ii, OpenCARP, and hiop with Ginkgo ensure the Reusable principle.

3.4 Summary

Ginkgo is a C++ high-performance sparse linear algebra library on several accelerators. We provide Ginkgo with good sustainability, general wide portability, high code quality, and many functionalities. Figure 3.1 shows the design of Ginkgo and Figure 3.6 shows the components reducing code duplication and the version history. Figure 3.7 shows the overview about the supported highlighted routines in Ginkgo among different backends. We have several advanced SpMV implementation and competitive performance [Anz+20b; Ali+21; Anz+20c], Krylov solver performance [Anz+22], advanced preconditioners [Tsa+22; Fle+21; Anz+19b], batched methods on matrix, solver, and preconditioner [NA23; Kas+23; Kas+22; Agg+22; Agg+21], GPU-resident direct solver [Świ+23], and algebraic Multigrid [TBA23a; TBA23b]. Besides the library infrastructure and optimization, we have provided an online interactive visualization tool GPE [Anz+19a] with continuous benchmarks. We extended Ginkgo's support to AMD and Intel GPUs during this dissertation. Thanks to Ginkgo's backend design, we can optimize the critical kernels such SpMV using the vendor native API in Chapter 4. We also extensively use Ginkgo's flexible linear operator and factories to achieve a flexible AMG in Chapter 5. Furthermore, we extend SpMV and AMG with mixed precision in Chapter 4 and Chapter 6.

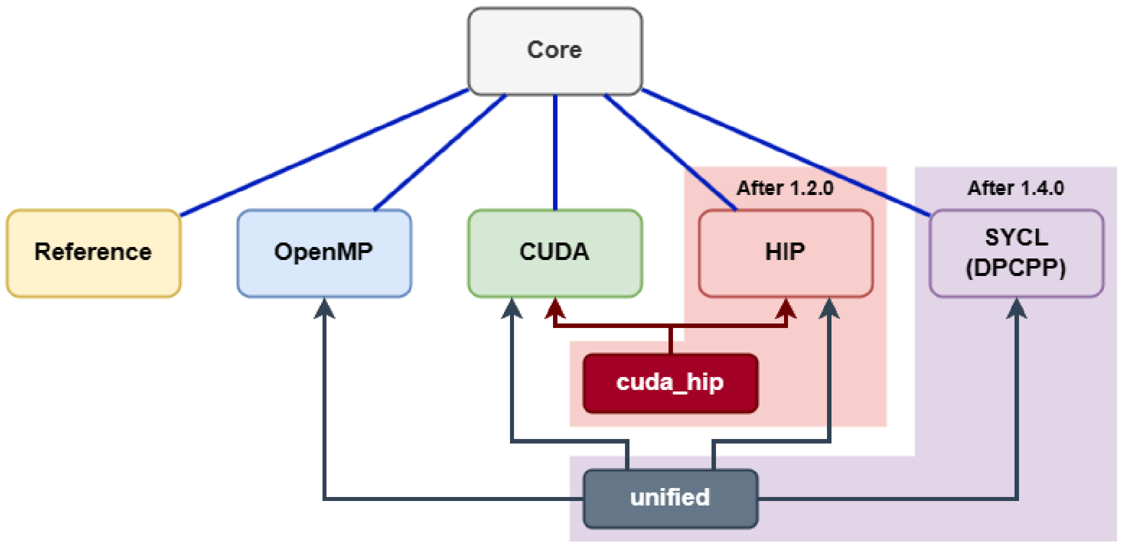
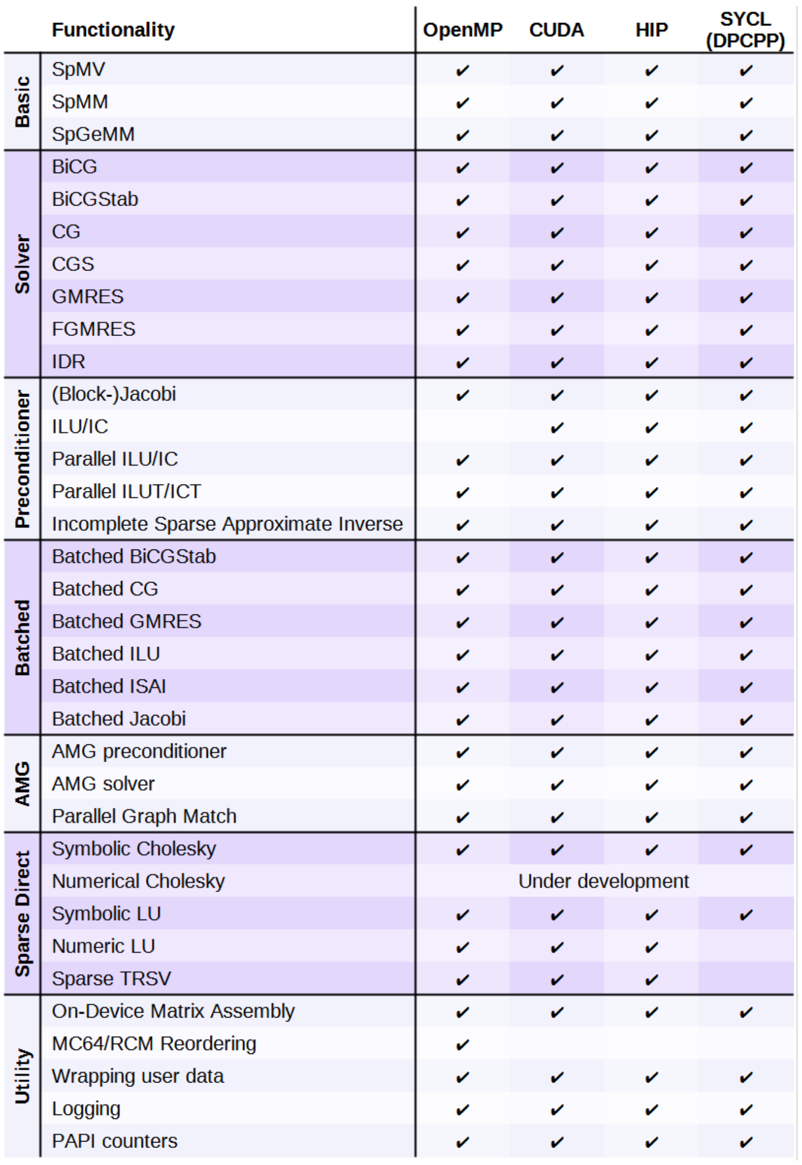


Figure 3.6: GINKGO library structure with more details and the version.



	Functionality	OpenMP	CUDA	HIP	SYCL (DPCPP)
Basic	SpMV	✔	✔	✔	✔
	SpMM	✔	✔	✔	✔
	SpGeMM	✔	✔	✔	✔
Solver	BiCG	✔	✔	✔	✔
	BiCGStab	✔	✔	✔	✔
	CG	✔	✔	✔	✔
	CGS	✔	✔	✔	✔
	GMRES	✔	✔	✔	✔
	FGMRES	✔	✔	✔	✔
	IDR	✔	✔	✔	✔
Preconditioner	(Block-)Jacobi	✔	✔	✔	✔
	ILU/IC		✔	✔	✔
	Parallel ILU/IC	✔	✔	✔	✔
	Parallel ILUT/ICT	✔	✔	✔	✔
	Incomplete Sparse Approximate Inverse	✔	✔	✔	✔
Batched	Batched BiCGStab	✔	✔	✔	✔
	Batched CG	✔	✔	✔	✔
	Batched GMRES	✔	✔	✔	✔
	Batched ILU	✔	✔	✔	✔
	Batched ISAI	✔	✔	✔	✔
	Batched Jacobi	✔	✔	✔	✔
AMG	AMG preconditioner	✔	✔	✔	✔
	AMG solver	✔	✔	✔	✔
	Parallel Graph Match	✔	✔	✔	✔
Sparse Direct	Symbolic Cholesky	✔	✔	✔	✔
	Numerical Cholesky	Under development			
	Symbolic LU	✔	✔	✔	✔
	Numeric LU	✔	✔	✔	
	Sparse TRSV	✔	✔	✔	
Utility	On-Device Matrix Assembly	✔	✔	✔	✔
	MC64/RCM Reordering	✔			
	Wrapping user data	✔	✔	✔	✔
	Logging	✔	✔	✔	✔
	PAPI counters	✔	✔	✔	✔

Figure 3.7: GINKGO highlighted supported functionalities among different accelerators.

4. SpMV

The sparse matrix-vector product (SpMV) is the backbone of sparse iterative methods and graph neural networks. SpMV is a heavily used and performance-critical operation in many applications, from Google's PageRank [LM12] to fluid flow simulations. SpMV is also a central kernel for the AMG method, and as this dissertation designs a platform portable mixed precision AMG algorithm, developing a high-performance portable SpMV is a natural first step. Operations on sparse matrices are usually memory-bound on all modern processors, including GPUs. Optimizing the SpMV kernel or finding a good way to compress the matrix for general purpose or specific applications, as well as supporting modern hardware is always an important research area in the high-performance computing (HPC) community.

In this chapter, we present the four SpMV formats (COO, CSR, ELL, and HYB introduced in Section 2.4) in Ginkgo and their realization on AMD, NVIDIA, and Intel GPUs. We also run performance experiments for each format using up to 2800 matrices from the SuiteSparse Collection [DH11] on AMD MI250x (1GCD), NVIDIA H100 (PCIE), Intel Max1550 (1tile). We also discuss some differences in the kernel design among these devices. All SpMV kernels are already available in the Ginkgo library introduced in Chapter 3.

The optimization of the SpMV kernels for GPU or CPU is still an area of active research[Dal+15; Hon+19; MG16]. Many of the algorithm developments increase the accumulation efficiency by using prefix-sum computations [MGG15] and intra-warp communication [Hon+11] on high-performance computing hardware. [Gro+16; Anz+20b] give a recent and comprehensive overview of SpMV research.

4.1 Design, Implementation, and Acceleration

We implement all SpMV kernels in the vendors' native languages as Chapter 3 mentioned: CUDA for NVIDIA GPUs, HIP for AMD GPUs, and SYCL(oneAPI) for Intel GPUs. Due to the different hardware characteristics and limits shown in Table 2.3, one parameter set for all devices is insufficient. We optimize these kernels' parameters or implementations for the distinct architectures to achieve higher performance. For example, we modify the load balance COO and CSR strategy for AMD and Intel GPU, which was previously discussed for NVIDIA architectures in [Anz+20b].

4.1.1 CSR

In the CSR SpMV, we can assign one thread to a row to compute the $A(i,*) \times b(*)$ results, which leads to a low occupancy of the GPU. To increase the utilization and use the warp technique, we assign a subwarp to each row for accumulation. Each subwarp will handle its own partial summation and then use subwarp reduction to the lead threads for writing the result to the global memory without using the more expensive atomicAdd. [TCA20] shows the CSR SpMV by using a single thread per row is not always slower than the classical CSR kernels. Assigning the same resource to those very imbalanced matrices may be inefficient in the end. Because some sparse matrices only contain short rows, some threads in warp will not have work to do if we always assign a full warp to each row. We select the subwarp/sub-sub-group size allowed depending on the devices. The allowed subwarp size is $2^k (0 \leq k \leq 5)$ for NVIDIA GPUs, $2^k (0 \leq k \leq 6)$ for AMD GPUs, (1, 16, 32) for the latest Intel GPU, and an additional 8 for the old Intel GPUs. We add 1 into the Intel GPUs allowed list because it can be considered a single thread without any sub-sub-group function support. Thus, we have sub-group sizes (1, 16, and 32) for Intel Max1550. As mentioned in Section 3.2.3, 1 is not the hardware-supported subgroup size. However, we consider it in the list because it does not need communication with the other threads. In the kernel generation, we select the closest subwarp size smaller or equal to the maximum number of nonzeros in rows, that is,

$$\text{subwarp size} = \max \{ k \leq \mathit{max\_row\_nnz} | k \in \text{device allowed size} \}$$

Depending on the problems, sparse matrices may be very pretty imbalanced, as shown in Figure 4.1 and Figure 4.2, which contains a few extremely long rows compared to the rest of the rows. When using the classical SpMV algorithm, which assigns a subwarp per row, the resources on short rows are idle after finishing their work because we still need to wait for the resources processing on the long rows as in Figure 4.1. Thus, the classical SpMV does not work well on imbalanced matrices. We instead use a load balance design for this situation by distributing the work according to the nonzeros in Figure 4.2. Load-balance SpMV assigns a similar nonzeros workload per resource. Each resource has a similar workload, but we instead need to use atomicAdd to avoid collisions because more than one resource may add partial results to the output vector at the same time. We will give more detail on a typical load-balance kernel implementation with the example of COO in Section 4.1.2.

Algorithm 11 GINKGO's classical CSR kernel.

```
1: Get $row$ = the row index
2: Compute $subrow$ = the step size to next row
3: Get $step\_size$ = the step size to the next element of value (subwarp size)
4: Get $subid$ = the local id of this subwarp.
5: Initialize value $c = 0$
6: for $row = row$ .. $\#rows$, $row+ = subrow$ do
7:     for $idx = row\_ptr[row]$ .. $row\_ptr[row + 1]$, $idx+ = step\_size$ do
8:         Compute $c+ = val[idx + subid] * b[col[idx + subid]]$
9:     end for
10:    Perform warp reduction of $c$ on the warp
11:    if the leading thread in subwarp then
12:        Write $c$ to the output vector
13:    end if
14: end for
```

Similarly to the COO load balance design of Section 4.1.2, we pre-generate the starting index for each warp to launch a load balance version for CSR. We also provide an automatic strategy to pick between the classical or load-balance strategy in Algorithm 12. There are two variables for the kernel decision. nnz_limit sets the condition for the limit of the total nonzero count, and row_len_limit sets the other condition for the limit of the maximum number of stored elements in a row. We select $nnz\_limit = 10^8$ and $row\_len\_limit = 768$ for AMD GPUs to select the load-balance strategy, $nnz\_limit = 10^6$ and $row\_len\_limit = 1024$ for NVIDIA GPUs, and $nnz\_limit = 3 \times 10^8$ and $row\_len\_limit = 25600$ for Intel GPUs. As we can see, the classical strategy is usually very competitive on Intel GPUs, which might related to the higher kernel overhead on Intel GPUs.

Algorithm 12 GINKGO's CSR strategy.

```
1: Compute $max\_row\_nnz$ = the maximal number of stored element per row.
2: if $\#nnz > nnz\_limit$ or $max\_row\_nnz > row\_len\_limit$ then
3:     Use load-balance CSR Kernel
4: else
5:     Use classical CSR Kernel
6: end if
```

4.1.2 COO

[FA17] introduced a load balanced COO SPMV by parallelizing across the nonzeros of a sparse matrix instead of parallelizing over the rows. In the nonzero-parallel distribution Figure 4.2, all subwarps have the same amount of nonzeros to process, and the coalesced access to memory is ensured. The `atomicAdd` functionality is needed to avoid race conditions when handling a row with several warps that simultaneously write partial results. Thus, the performance of atomic operations on the output vector is critical for the load-balance SPMV. For threads of the same warp, a warp-local segmented scan can be used for the local reduction, thereby reducing the number of atomic collisions in the output vector. We detail the algorithm in Algorithm 13 which uses the CUDA terms from Table 2.1.

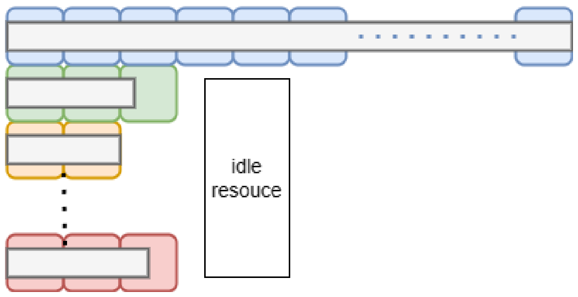


Figure 4.1: Row (Warp) parallel scheme (classical SpMV) on imbalanced matrix. A subwarp handles the blocks with the same color.

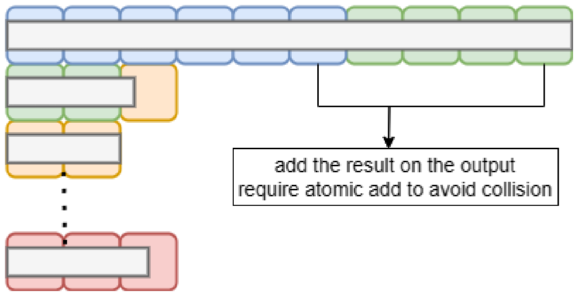

add the result on the output
require atomic add to avoid collision

Figure 4.2: Nonzero parallel scheme (load balance SpMV) on imbalanced matrix. A subwarp handles the blocks with the same color.

The parameter ω in Equations (4.1) to (4.3) controls the level of oversubscription. More active threads can hide the latency of data access and the atomicAdd [FA17; TCA20], but they also increase the number of atomicAdd collisions and the overhead from context switching. From the experimental results on all real matrices from the SuiteSparse Matrix Collection, we choose the following parameter configuration for ω:

$$\omega_{\text{NVIDIA}} = \begin{cases} 8 & (n_z < 2 \cdot 10^5), \\ 32 & (2 \cdot 10^5 \leq n_z < 2 \cdot 10^6), \\ 128 & (2 \cdot 10^6 \leq n_z < 2 \cdot 10^7), \\ 512 & (2 \cdot 10^7 \leq n_z < 2 \cdot 10^8), \\ 2048 & (2 \cdot 10^8 \leq n_z), \end{cases} \tag{4.1}$$

$$\omega_{\text{AMD}} = \begin{cases} 2 & (n_z < 10^5), \\ 8 & (10^5 \leq n_z < 10^7). \\ 32 & (10^7 \leq n_z) \end{cases} \tag{4.2}$$

$$\omega_{\text{Intel}} = \begin{cases} 8 & (n_z < 2 \cdot 10^7), \\ 32 & (2 \cdot 10^7 \leq n_z < 2 \cdot 10^8). \\ 256 & (2 \cdot 10^8 \leq n_z) \end{cases} \tag{4.3}$$

4.1.3 ELL

[Anz+20b] applies the same concepts of classical CSR to ELL, which accelerates kernels by assigning multiple threads to each row. Moreover, it uses "early stopping" to free resources if threads operate on the unused padding added to the ELL structure. Instead, we realize non-coalesced global memory access has a larger impact than the atomicAdd usage to shared memory for AMD architectures in [TCA20]. The original

Algorithm 13 Load-balancing COO kernel algorithm [FA17]

```
 1: Get $ind$ = index of the first element to be processed by this thread
 2: Get $current\_row = row\_idx[ind]$.
 3: Compute the first value $c = A[ind] \times x[col\_idx[ind]]$
 4: for i = 0 .. $nz\_per\_warp$; $i+ = warp\_size$ do
 5:     Compute $next\_row$, row index of the next element to be processed
 6:     if any thread in the warp's $next\_row$ != $current\_row$ or it is the final
        iteration then
 7:         Compute the segmented scan according to $current\_row$.
 8:         if first thread in segment then
 9:             atomicAdd $c$ on output vector by the first entry of each segment
10:         end if
11:         Reinitialize $c = 0$
12:     end if
13:     Get the next index $ind$
14:     Compute $c+ = A[ind] \times x[col\_idx[ind]]$
15:     Update $current\_row$ to $next\_row$
16: end for
```

design [Anz+20b] uses threads of the same subwarp to handle one row like classical CSR, which results in adjacent threads always reading non-coalesced global memory due to the column-major storage for ELL. GINKGO rearranges the algorithm by assigning the threads of the same warp to handle one column of ELL storage to achieve coalesced memory access. However, several warps will work on the same row, leading to a race condition. We first use atomicAdd on the shared memory, not the global memory, if there is only one thread block on the same row. This creates a hierarchical reduction using shared memory atomics and global memory atomics. Figure 4.3 visualizes the different memory access strategies. From the paper, we conclude that both versions perform well on NVIDIA GPUs, but only the coalesced memory works better for AMD GPUs. We only keep the coalesced memory version for all GPU backends for maintenance. We detail the coalesced memory version in Algorithm 14 and the parameter settings in Algorithm 15

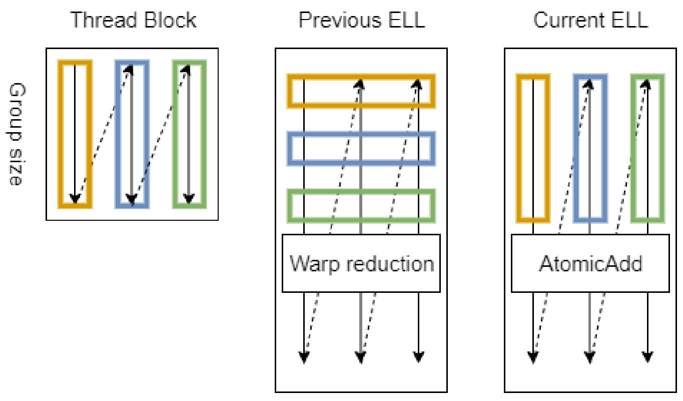


Figure 4.3: Comparison of the memory access for different ELL SPMV kernels in [Anz+20b].

Algorithm 14 GINKGO's ELL SPMV kernel.

```
1: Initialize Value $c = 0$
2: Compute $row$ = the row idx
3: Compute $y$ = the start index of row
4: Compute $step\_size$ = the step size to the next element
5: Initialize shared memory $data$
6: for $idx = y$ .. $max\_row\_nnz$, $idx += step\_size$ do
7:     Compute $ind$ = index of this element in the ELL format
8:     if $A(row, colidx[ind])$ is unused then
9:         break
10:    end if
11:    Perform local operation $c += A(row, colidx[ind]) * x[colidx[ind]]$
12: end for
13: Perform atomicAdd $c$ to shared memory $data[threadIdx.x]$
14: if thread 0 in group then
15:     atomicAdd $data[threadIdx.x]$ on the output vector
16: end if
```

Algorithm 15 GINKGO's automatic ELL kernel configuration.

```
1: Initialize num_group = 1
2: Initialize nblock_per_row = 1
3: Compute ell_ncols = maximum number of nonzero elements per row
4: Get nwarps = total number of warps available on the GPU
5: if ell_ncols / nrows > $1e-2$ then
6:     Compute num_group = $min(16, 2^{ceil(log_2(ell\_ncols))})$
7:     if num_group == 16 then
8:         Compute nblock_per_row = $max(min(ell\_ncols/16, nwarps/nrows), 1)$
9:     end if
10: end if
```

Algorithm 14 shows the ELL SPMV kernels in GINKGO. Each thread accumulates the result into the register by Line 11 until the unused part or the end of rows, add the result in the shared memory by atomic operation by Line 13 as many threads of the same block may process on the same row. In the end, only one thread per row in a block adds the result from shared memory to global memory atomically, as many blocks may process on the same row. Some parameters in Algorithm 14 like num_group and nblock_per_row need to be decided by Algorithm 15. The strategy of Algorithm 15 is to try to increase the occupancy of GPUs when targeting short-and-wide matrices, so it may have many thread blocks on the same row. The kernel still keeps the early-stopping technique in lines 8-10 introduced in [Anz+20b]. After accumulating the partial result, each thread will perform an atomicAdd to the shared memory. Then, the leading thread will perform atomicAdd to the global memory of the output vector in the end.

In CUDA and HIP, we consider the available subwarp size for the number of working threads for each row in the warp. Thus, we first only consider the sizes (1, 16, 32) as the virtual sub-sub-group for the SYCL backend during the porting effort. However, we only use the shared memory without the communication in the sub-group in

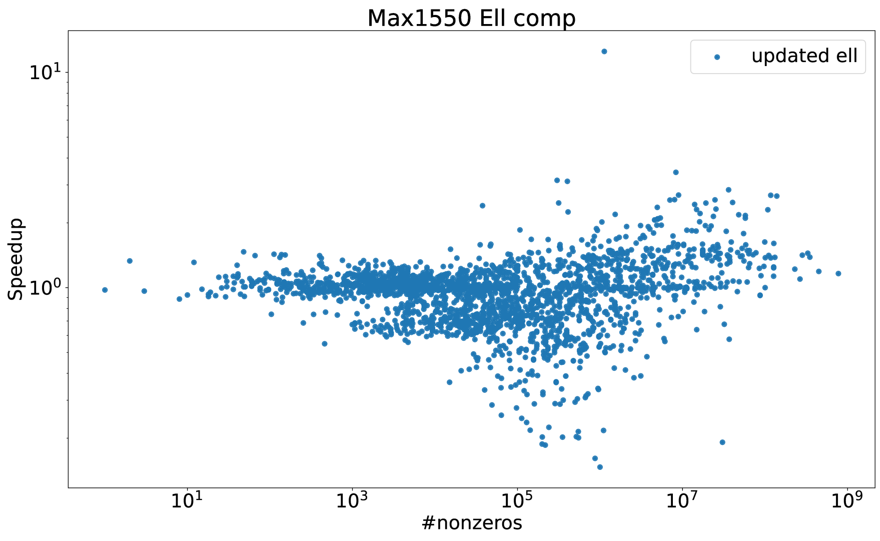


Figure 4.4: ELL performance comparison on Max1550 (1tile): ELL with available sub-group size (1, 2, 4, 8, 16, 32) against (1, 16, 32)

Algorithm 14. We do not need to worry about the sub-group size, so we also consider the options among (1, 2, 4, 8, 16, 32) for the number of threads for a row in the sub-group, which leads to the performance comparison in Figure 4.4. ELL is also an important component in HYB. The updated implementation shows apparent speedup in Figure 4.5. If the hybrid is the required format in the application, the ELL with more options gives a clear benefit for HYB format.

4.1.4 HYBRID(HYB)

The HYB format is a combination of COO and ELL, which tries to get both benefits from COO and ELL. HYB splits the matrix to store the regular part in the ELL format and then put the rest of the value (irregular part) in the COO format. From the paper[Anz+20b], we have derived a strategy based on the nonzeros-per-row distribution and the ratio between the maximum nonzeros-per-row and the number of rows. For R being the set of the nonzeros-per-row values, we define the function Q_R and F_R:

$$Q_R(x) := \min \{t \in \mathbb{N} \mid x < F_R(t)\}, \qquad F_R(t) := \frac{|\{r \in R \mid r \le t\}|}{|R|}.$$

The method introduced in [Anz+20b] uses $n\%$-quantile of the nonzeros of the row, which is $Q_R(n\%)$ here. We can also calculate the minimal storage for the HYB format by selecting

$$n\% = \left\lfloor \frac{\#rows \times sizeof(index)}{sizeof(value) + 2 \times sizeof(index)} + 1 \right\rfloor$$

with the size of value and index type. When using double (64-bit) for value type and int(32-bit) for index type, the hybrid25 provides minimal storage. We also notice that hybrid with minimal storage does not imply the best performance. In Ginkgo, we also have another limitation such that the ELL part can only allow up to $\#rows * 0.0001$ for columns. We consider the resulting strategy “hybridlimit{n}” and select hybridlimit33 (label “HYB”) as our default strategy.

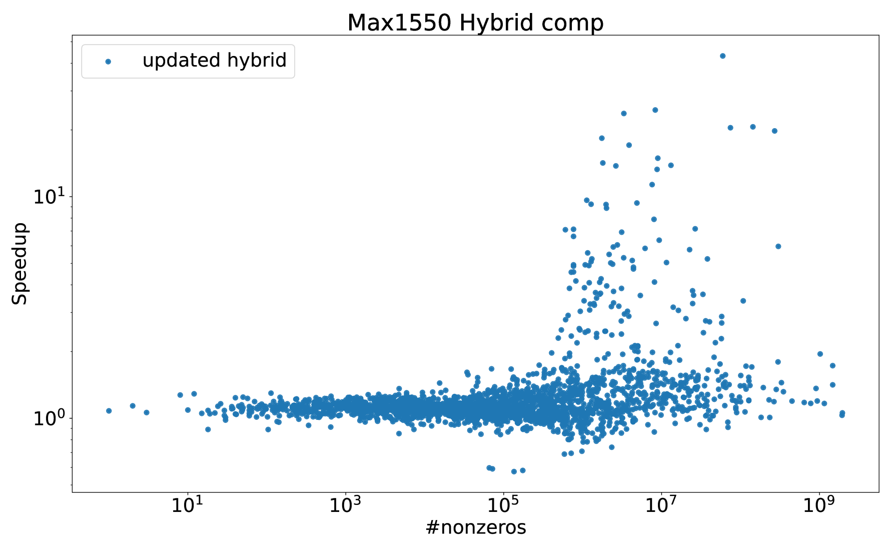


Figure 4.5: HYB performance comparison on Max1550 (1tile): The component ELL with available sub-group size (1, 2, 4, 8, 16, 32) against (1, 16, 32)

4.2 Mixed SpMV Support

In Ginkgo, we support bfloat16, half, single, and double precision SpMV and the corresponding complex versions. The SpMV implementation uses the accessors introduced in [GAQ23], which is a helper that can read the data stored in the global memory to the arithmetic format in the register for operations directly without an explicit copy. In Ginkgo, the arithmetic precision is chosen as the precision can losslessly store input, output, and matrix precision. We choose the single precision as the arithmetic precision between half-precision and bfloat16 precision. Figure 4.6 visualizes the difference between implicit and explicit precision conversion with the example

$$Y_{\text{single precision}} = A_{\text{half precision}} X_{\text{double precision}}$$

The explicit precision conversion must copy A from half precision to double precision before operations and copy Y from double precision to single precision in global memory when writing results. In Section 2.2, we established that the small arithmetic intensity characterizes SpMV as a memory-bound operation. However, mixed precision SpMV reads the data in low precision without requiring an explicit copy of the data for high precision arithmetic, therewith reducing the memory footprint and combining faster execution with higher accuracy than low precision SpMV.

4.2.1 Accessor

The idea of the accessor is to decouple the memory access out of the implementation. The idea in Figure 4.6 to convert the data on the fly is not easy in practice when considering the efficiency and the capability for different precision. We may rely on the implicit type conversion by compilers during the arithmetic operation if we only consider single and double precision. It is not trivial when we consider bfloat16 and half precision. As the bfloat16 and half precision are still new to programming language, compilers may not deal with the arithmetic operation with the other types well. Also, some compiler does not allow the arithmetic operation between two

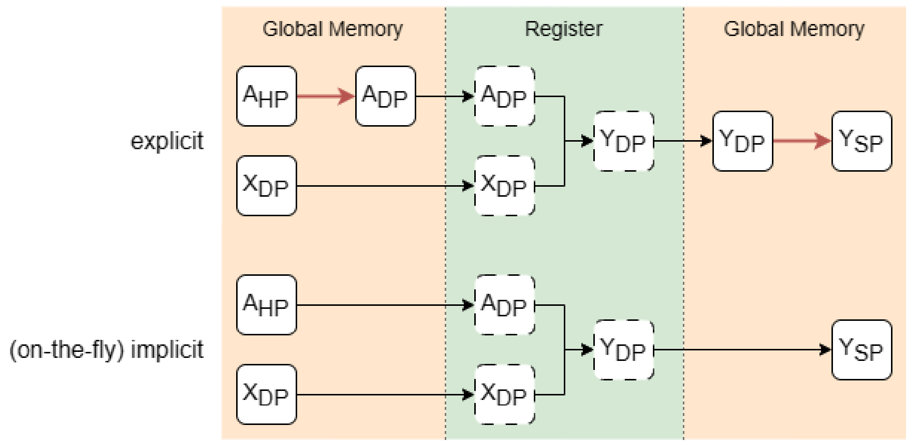
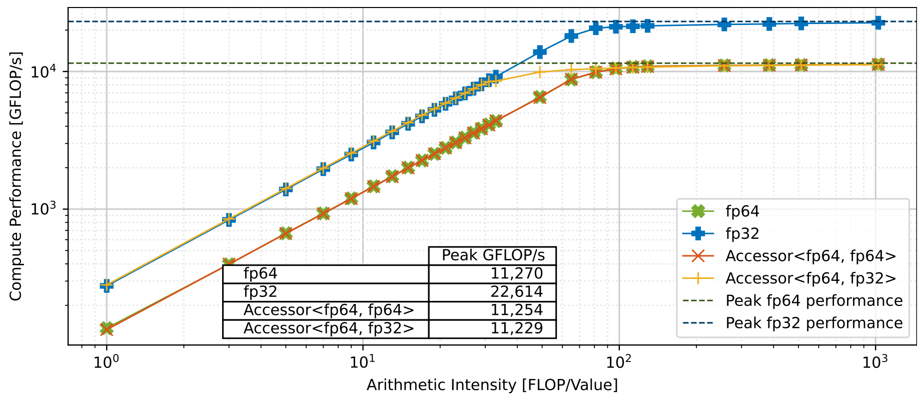


Figure 4.6: Implicit vs explicit precision conversion from memory with the example $Y_{\text{single precision}} = A_{\text{half precision}} X_{\text{double precision}}$. Explicit precision conversion needs additional copies to prepare the objects with arithmetic precision. The red arrow indicates the additional conversion in global memory. However, implicit precision conversion can directly get the arithmetic precision in the register.

different precisions of complex numbers. Suppose we rely on the explicit type casting (such as `static_cast<target>(source)` from C++), the casting usage increases the maintenance difficulty. Thus, we need a way to decouple the precision conversion out of the kernel implementation, and it must be efficient and support different precision conversions.

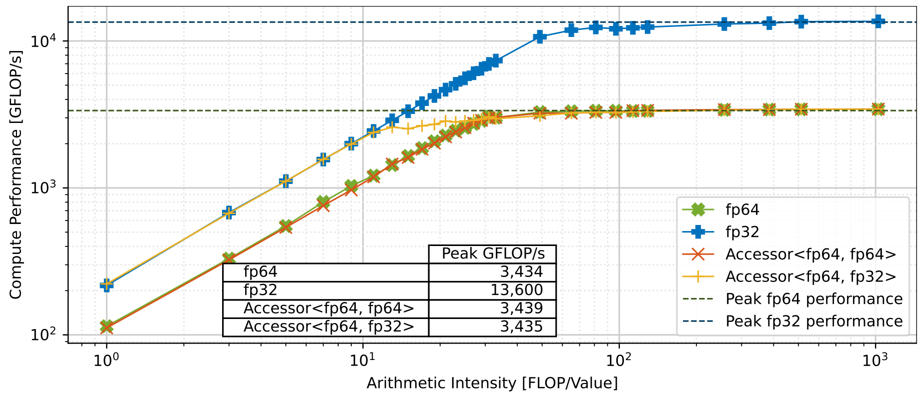
Figures 4.7 to 4.9 reports the performance of the memory accessor in a roofline analysis on AMD MI100, AMD RadeonVII, and NVIDIA V100 GPUs from [GAQ23]. They use FLOP/{the number of values} to demonstrate the performance based on the number, not the size of data. We can recognize the difference between using different precision on the same number of values (i.e., the same x-axis value in the figures). They use fp32 for single precision and fp64 for double precision in [GAQ23]. The horizontal dashed line indicates the theoretical performance peaks for single precision (fp32) and double precision (fp64). When the kernels are in the memory-bound area (i.e., before reaching the dashed line), the single precision implementation reaches the 2x speedup of the double precision one because the single precision requests only half the amount of memory of the double precision. The single precision still reaches 2x speedup of the double precision for compute-bound kernels in Figures 4.7 and 4.9 because the server GPUs usually make single precision performance is 2x speedup of the double precision performance as in Table 2.3. The single precision shows 4x speedup than the single precision performance on AMD RadeonVII in Figure 4.8 because the consumer GPUs usually have less double precision capability as in Table 2.3.

[GAQ23] uses Figures 4.7 to 4.9 to show the accessor efficiency. `accessor<fp64, fp64>` reads the double-precision data and processes the data in double precision as normal. It sticks with `fp64` in Figures 4.7 to 4.9, so accessor does not introduce overhead when we do the same operation. `accessor<fp64, fp32>` reads the single-precision data but processes the data in double precision. When the kernel is memory-bound, `accessor<fp64, fp32>` shows the same performance as `fp32`. That is, we can indeed get the performance benefit from reading the single-precision



	Peak GFLOP/s
fp64	11,270
fp32	22,614
Accessor<fp64, fp64>	11,254
Accessor<fp64, fp32>	11,229

Figure 4.7: The roofline performance of accessor on AMD MI100 GPU from [GAQ23]. The Arithmetic Intensity uses FLOP/{the number of values}.



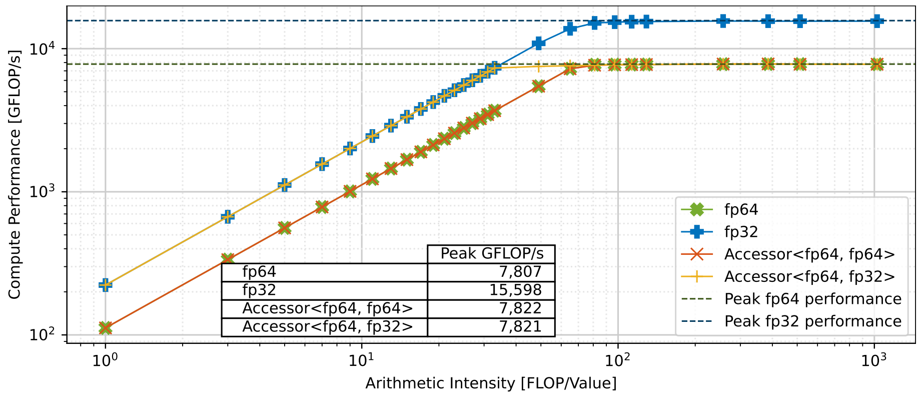
	Peak GFLOP/s
fp64	3,434
fp32	13,600
Accessor<fp64, fp64>	3,439
Accessor<fp64, fp32>	3,435

Figure 4.8: The roofline performance of the accessor on AMD RadeonVII GPU from [GAQ23]. The Arithmetic Intensity uses FLOP/{the number of values}.

data. When the kernel is compute-bound, `accessor<fp64, fp32>` can only reach the `fp64` peak, which shows the same thing as roofline model Section 2.2. We compute the data in double precision, so the double-precision performance peak bounds the compute-bound kernels. The accessor helps us to decouple the memory and computing.

4.2.2 Use Accessor in SpMV

The accessor helps make the code clear and keep the performance. As we add the bfloat16 and half precision formats into the Ginkgo precision support, we can not directly rely on the implicit type conversion, which some compilers do not support it with half and bfloat16 precision. We show the mixed-precision SpMV kernel implementation with the `static_cast` in Listing 4.1 and the accessor in Listing 4.2. In Listing 4.1, developers must remember to cast the data correctly to ensure that the mixed-precision operation uses arithmetic precision. However, in Listing 4.2, developers only need to wrap the data with the accessor and then access the data through the accessor, which is almost like the original SpMV implementation. The decoupling helps us easily ensure that the code does the correct mixed-precision without worrying about the casting. Using casting is difficult when implementing optimized SpMV kernels and advanced algorithms. Based on the accessor, we have implemented ELL and CSR mixed-precision SpMV in Ginkgo.



	Peak GFLOP/s
fp64	7,807
fp32	15,598
Accessor<fp64, fp64>	7,822
Accessor<fp64, fp32>	7,821

Figure 4.9: The roofline performance of the accessor on an NVIDIA V100 GPU from [GAQ23]. The Arithmetic Intensity uses FLOP/{the number of values}.

```
// using static_cast
template <typename MatrixPrec, typename InputPrec, typename OutputPrec>
void csr_spmv(MatrixPrec* A, InputPrec* X, OutputPrec* Y) {
    // decide the arithmetic precision
    using ArithmeticPrec = typename precision<MatrixPrec, InputPrec, OutputPrec>::
        type;
    for (row in A) {
        // Keep the accumulation in arithmetic precision
        ArithmeticPrec temp(0.0);
        for (int i = A_row_ptr[row]; i < A_row_ptr[row+1]; i++) {
            auto col = A_col_idx[i];
            // cast the data to arithmetic precision
            temp += static_cast<ArithmeticPrec>(A_val[i]) * static_cast<
                ArithmeticPrec>(X[col]);
        }
        // cast back to output precision
        Y[row] = static_cast<OutputPrec>(temp);
    }
}
```

Listing 4.1: The mixed-precision CSR SpMV using `static_cast`

```
// using accessor
template <typename MatrixPrec, typename InputPrec, typename OutputPrec>
void csr_spmv(MatrixPrec* A, InputPrec* X, OutputPrec* Y) {
    // decide the arithmetic precision
    using ArithmeticPrec = typename precision<MatrixPrec, InputPrec, OutputPrec>::
        type;
    // Wrap the data with the accessor
    A_acc = accessor<ArithmeticPrec>(A);
    X_acc = accessor<ArithmeticPrec>(X);
    Y_acc = accessor<ArithmeticPrec>(Y);
    for (row in A) {
        // Keep the accumulation in arithmetic precision
        ArithmeticPrec temp(0.0);
        for (int i = A_row_ptr[row]; i < A_row_ptr[row+1]; i++) {
            auto col = A_col_idx[i];
            // accessor gives the arithmetic-precision data
            temp += A_acc_val(i) * X_acc(col);
        }
        // accessor accept the arithmetic-precision data
        Y_acc(row) = temp;
    }
}
```

Listing 4.2: The mixed-precision CSR SpMV using the accessor

4.2.3 Capability for Mixed-Precision SpMV

For convenience, we introduce the precision format notation in Table 4.1. We focus

DP	IEEE 754 double precision (64-bits)
SP	IEEE 754 single precision (32-bits)
HP	IEEE 754 half precision (16-bits)
BF	Google bfloat16 precision (16-bits)
I8	8-bit integer
I32	32-bit integer
CPX<V>	complex number with V precision

Table 4.1: Precision format notation.

on the mixed-precision operations on CSR because CSR is the most widely used in different algorithms and the commonly supported format in vendors' libraries. We list the precision support of CSR SpMV from Ginkgo, AMD rocSPARSE, Intel oneMKL, and NVIDIA cuSPARSE in Table 4.2. As Table 4.2 lists, cuSPARSE and rocSPARSE support some mixed-precision SpMV and the complex versions, but oneMKL only supports real uniform precision SpMV. cuSPARSE always requires the same precision for the matrix A, the input vectors X, and the specific arithmetic precision. rocSPARSE may require the same precision for A and X, or for X and Y. Although hipSPARSE provides the interface for cuSPARSE and rocSPARSE, users still need to adapt the variable for different platforms due to an inconsistency between cuSPARSE and rocSPARSE.

In contrast, Ginkgo's design allows any combination of precision of the matrix A, the input vectors X, and the output vectors Y on these platforms. The arithmetic precision is automatically decided by Ginkgo such that users do not need to check what arithmetic type is used for the current precision combination. For the uniform precision operations with 16-bit floating point numbers, cuSPARSE and rocSPARSE always use 32-bit floating point format as arithmetic precision for computation, but Ginkgo still uses 16-bit floating point like the other uniform precision operations. Thus, we mark the entry with 2 for cuSPARSE supporting 16-bit operations in mixed section but 3 for ginkgo supporting 16-bit operation in uniform precision section in Table 4.2. rocSPARSE and cuSPARSE support the 8-bit integer operations additionally. Ginkgo provides the complete set of mixed precision without any additional copy routine and overhead, as well as the corresponding complex number version.

In the advanced apply signature from Chapter 3, there are three precision formats that can be selected in A->apply(α, X, β, Y). Matrix precision is for A, input precision is for vector X, and output precision is for vector Y. The scalar α uses the same precision as A and β uses the same precision as Y. This is slightly different from the NVIDIA cuSPARSE and the AMD rocSPARSE design, which use arithmetic precision for scalar parameters. Our scalar only relies on the storage type of corresponding components. The benefit of this design is that we do not need to know everything beforehand for the decision of the scalar storage. We compare the precision format support of the mixed-precision SpMV with the formula $Y = AX$ in Ginkgo and the other three vendors' libraries in Table 4.2. To reduce the table complexity, we only consider input, output, and matrix precision without the scalar precision because these known factors always decide scalar precision.

	Ginkgo	NVIDIA	AMD	Intel
uniform precision				
HP	√	(√)[1]		
BF	√	(√)[1]		
SP	√	√	√	√
DP	√	√	√	√
CPX<HP>	√	(√)[1]		
CPX<BF>	√	(√)[1]		
CPX<SP>	√	√	√	√[2]
CPX<DP>	√	√	√	√[2]
real complex (matrix / vectors)				
SP / CPX<SP>	√	√	√	
DP / CPX<DP>	√	√	√	
mixed (A / X / Y / arithmetic)				
I8 / I8 / I32 / I32		√	√	
I8 / I8 / SP / SP		√	√	
HP / HP / SP / SP	√	√		
BF / BF / SP / SP	√	√		
HP / HP / HP / SP	(√)[3]	√		
BF / BF / BF / SP	(√)[3]	√		
CPX<HP> / CPX<HP> / CPX<HP> / CPX<SP>	(√)[3]	√(deprecated)		
CPX<BF> / CPX<BF> / CPX<BF> / CPX<SP>	(√)[3]	√(deprecated)		
SP / DP / DP / DP	√		√	
CPX<SP> / CPX<DP> / CPX<DP> / CPX<DP>	√		√	
<type_set> / <type_set> / <type_set> / auto (64 kinds)	√			
<type_set> / CPX<type_set> / CPX<type_set> / auto (64 kinds) (real complex)	√			
CPX<type_set> / CPX<type_set> / CPX<type_set> / auto (64 kinds)	√			

Table 4.2: The supported mixed precision SpMV ($Y = AX$) from Ginkgo and vendors' libraries. Table 4.1 shows the precision format notation. The type_set is any precision format among bfloat16(BF), half(HF), single(SP), and double(DP) precision.

Version information: NVIDIA - cuSPARSE 12.3; AMD - rocSPARSE 6.0.0; Intel - oneMKL 2024.0

[1]cuSPARSE supports these precisions with single precision for computation in mixed precision.

[2]oneMKL starts support complex type from 2024.0

[3]Ginkgo supports these with the same precision for computation in uniform precision

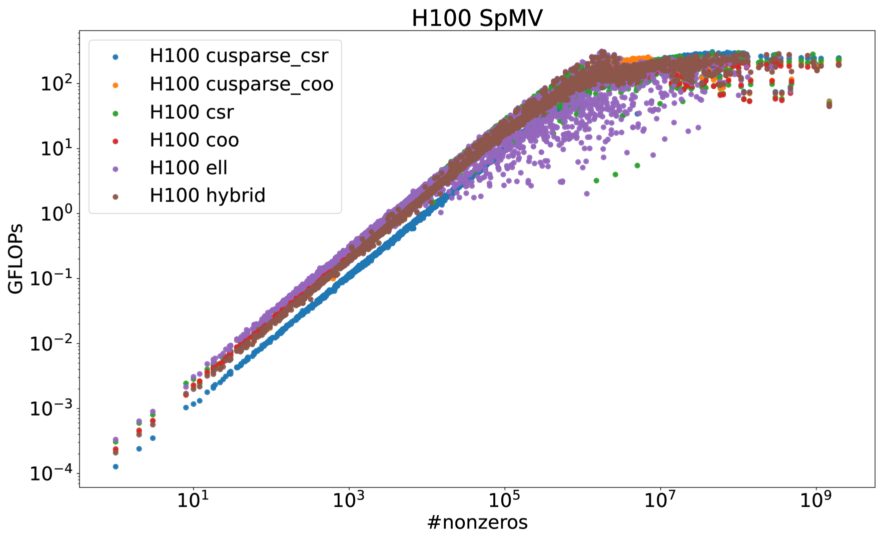


Figure 4.10: SpMV performance comparison on H100(PCIE)

4.3 Experiments

We evaluate CSR and COO from the vendor libraries, and CSR, COO, ELL, HYB from GINKGO library on three vendors' GPUs. The details of the testing environments are listed in Section 2.7. We perform 2 iterations for warmup and 10 iterations for performance measurements. We take the median value from these 10 iterations as the results. Intel oneMKL only supports the CSR format, so we can not report the oneMKL COO format on Intel GPUs. We test SPMV with considered formats on all real matrices from the SuiteSparse Matrix Collection [DH11], which contains around 2,800 matrices. We ignore some matrices on certain formats because these matrices can not be stored in the corresponding GPU memory with certain formats. For example, the imbalanced matrix with long rows leads to ELL requiring a massive amount of memory on GPU.

4.3.1 Double Precision SpMV Performance

We consider IEEE double precision SpMV first in the following discussion. The computation of GFLOP/s is under the assumption that the number of flops is $2nnz$, where nnz is the number of stored elements of the matrix data.

We collect the performance data in Figure 4.10 on H100. ELL achieves good performance for matrices with few nonzeros, but starts to have some slower cases than the others when the number of nonzeros is larger than 10^4 due to imbalanced matrices. Although COO can not reach the highest performance, COO achieves good performance for all matrices considered. cuSPARSE CSR shows some overhead when the number of nonzeros is less than 10^6 because the cuSPARSE Generic API may need additional work to wrap the data to its framework. The cuSPARSE COO has less overhead than the cuSPARSE CSR. HYB shows a more stable performance than ELL, which prevents the imbalanced matrices from using ELL entirely. GINKGO's CSR and COO achieve a competitive performance against cuSPARSE in large cases with less overhead in small cases.

As discussed in Section 2.8, we use a performance profile to plot the overview among this dataset on H100 in Figure 4.11. Both GINKGO's CSR and ELL win 30% of

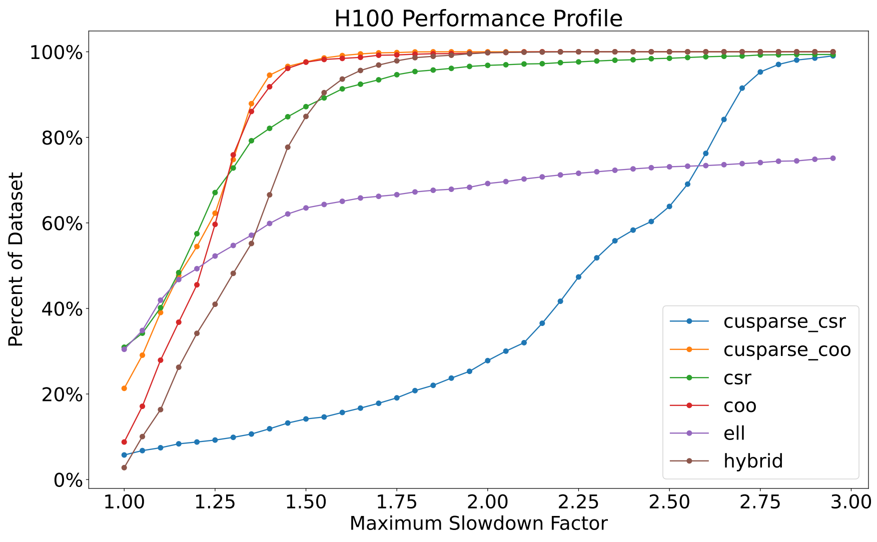


Figure 4.11: SpMV performance profile on H100(PCIE)

problems (at x = 1), which is a higher rate than the others. When we allow a little slowdown factor, Ginkgo's COO and cuSPARSE COO can solve almost all problems within 1.5x slowdown allowance, and HYB can solve almost all problems within 1.75x slowdown allowance. Even if we allow 3x slowdown, ELL can only solve 60% problems due to the storage limit. CSR has a similar trend with COO but loses a little generalization.

Figure 4.12 shows the performance results on AMD MI250X(1GCD). Overall, a clear trend is that Ginkgo outperforms AMD hipSPARSE, which may be from using unsafe atomic instruction. HIP provides a compiler option to turn on the hardware-based floating point atomic operation with less guarantee, which is correct only on coarse-grained memory. Although unsafe atomic operation contains unsafe in the name, we use it correctly and safely in Ginkgo. We use `hipMalloc`, which is coarse-grained memory, such that Ginkgo enables this instruction safely and correctly. ELL on MI250X has a similar story to H100, which has slower performance with the imbalanced matrices. hipSPARSE CSR encounters a similar performance drop for the imbalanced matrices. With the load balance implementation, Ginkgo's CSR provides more stable performance than hipSPARSE. HYB still shows a performance between COO and ELL.

Figure 4.13 uses the performance profile for the SpMV performance overview on MI250X(1GCD). As Figure 4.11, Figure 4.13 also shows the Ginkgo's SpMV kernels win a high ratio of the dataset. Ginkgo's CSR wins more cases and it is a more general solution than Ginkgo's COO, which is unlike H100 in Figure 4.11. ELL solves around 70% of problems within 3x slowdown allowance. Although HYB is not the fastest among all formats, it can solve more problems than COO and ELL when we allow some slowdown. HYB shows a similar generality with Ginkgo's CSR.

Figure 4.14 collects the performance data on Intel Max1550 (1tile). Unlike the other platforms, COO and HYB are slower than the other formats in Ginkgo. COO or load balanced CSR needs to initialize the vectors by zero first and then apply atomic operations on the vectors, and HYB needs to use ELL kernels first and then use COO kernels. The kernels generated for Intel GPUs may have a higher overhead

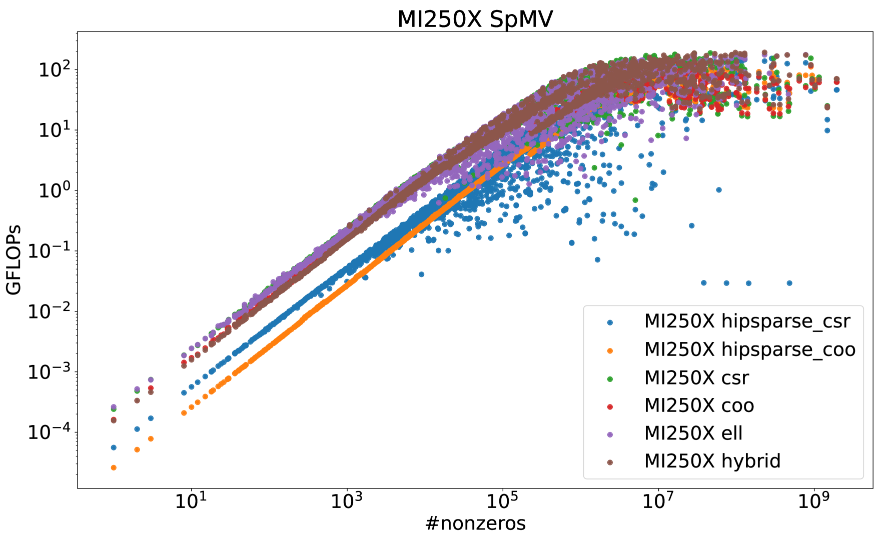


Figure 4.12: SpMV performance comparison on MI250X(1GCD)

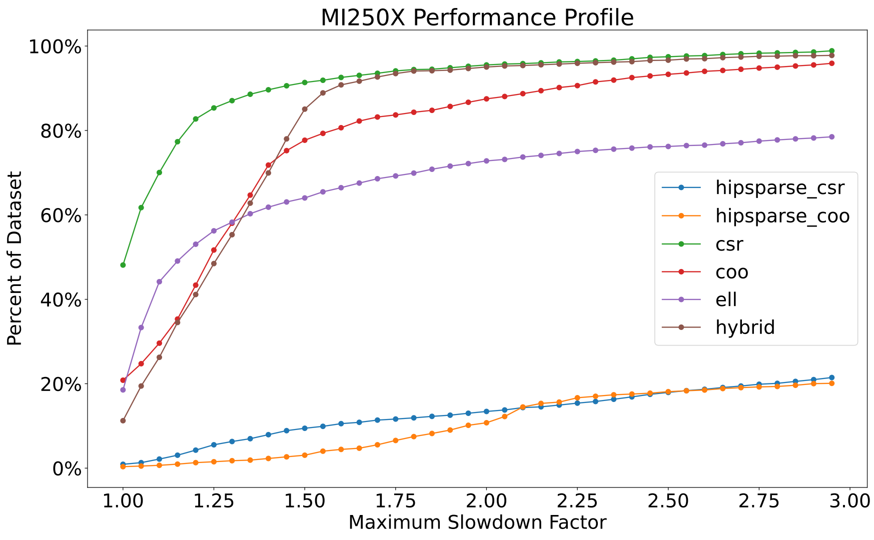


Figure 4.13: SpMV performance profile on MI250X(1GCD)

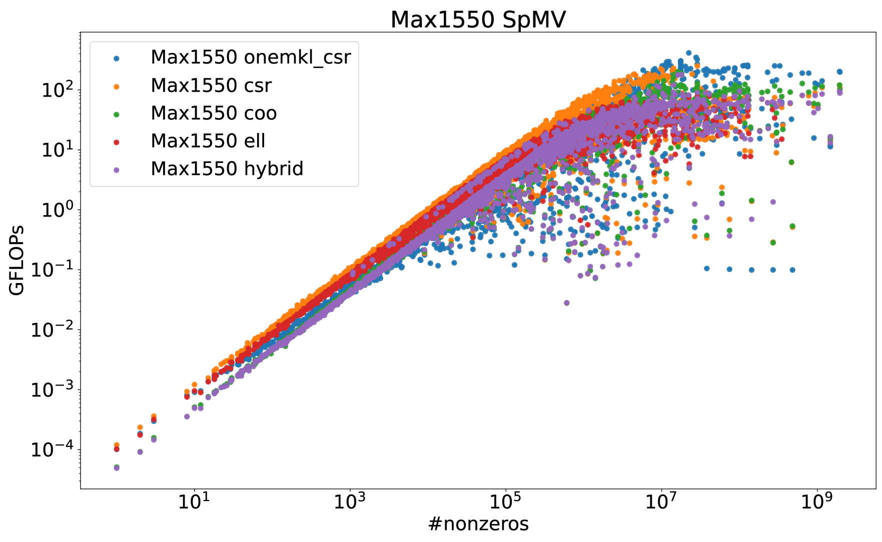


Figure 4.14: SpMV performance comparison on Max1550 (1tile). oneMKL only supports CSR format.

than the others such that these SpMV composed of two kernels are slow. With that observation, Ginkgo CSR usually selects classical CSR, not load balanced CSR. Although we hit a similar peak on the cases that contain around 10^7 nonzeros, Ginkgo CSR is slightly slower than oneMKL in extremely large cases. We know the performance can be achieved by using one thread per row in classical CSR. Without setting complex rules, we decided to provide a more stable performance, which provides higher performance than oneMKL for those data points in the right-bottom part. When the kernel overhead is lower in the future, the load balance should cover this region to provide a similar peak performance. Max1550 only provides 1, 16, and 32 for sub-group size, which we mentioned in Section 4.1.1, so we do not have a good option between 1 and 16, unlike the other backends.

Figure 4.15 shows the performance profile among all cases on Max1550 (1tile). Ginkgo's CSR is always at the top of Figure 4.15. As discussed in the previous paragraph, the other formats in Ginkgo containing more than one kernel suffer from kernel overhead. Without considering Ginkgo's CSR, ELL solves more problems within 1.25x - 2.00x slowdown allowance, but COO solves most of the problems after 2x slowdown allowance.

4.3.2 Half Precision SpMV Performance

In the following, we focus on the half-precision CSR, which Ginkgo multigrid mainly uses in Chapter 6. We have two variants for classical CSR with half precision. One is Algorithm 11, which was already introduced in the previous section, and the other is the packed version Algorithm 16. NVIDIA GPUs prefer getting 128 bytes of data, which is 4 bytes per thread in a warp. However, if we only read one half-precision value per thread, we may not fully utilize the GPU bandwidth. Thus, we designed a packed version listed in Algorithm 16 that always accesses two half-precision values such that the compiler may optimize the code. We collect the problem-wise comparison between two versions of CSR in Figure 4.16 on H100, Figure 4.17 on MI250X(1GCD), and Figure 4.18 on Max1550 (1tile). The packed version on H100

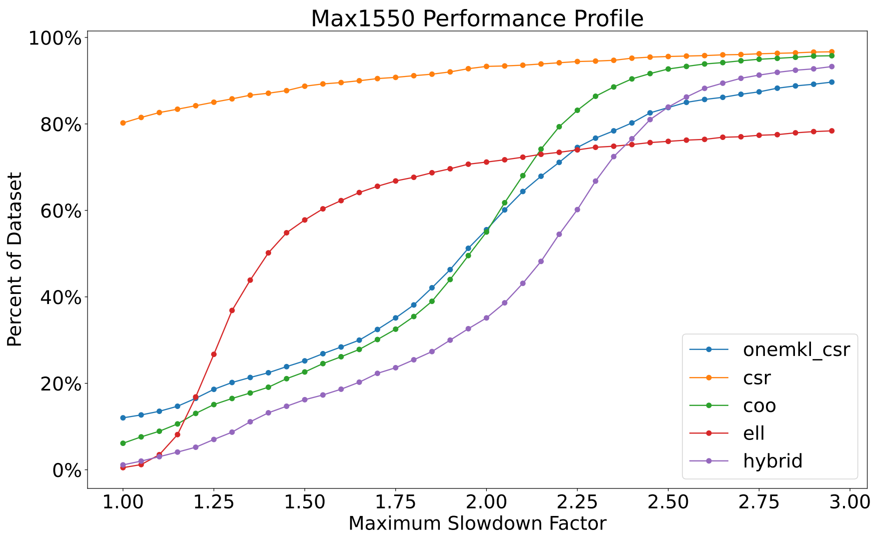


Figure 4.15: SpMV performance profile on Max1550 (1tile)

shows around 2x speedup but around 0.8x - 0.9x slowdown compared to the original version. Including this slowdown effect, GINKGO still shows better performance in small cases and competitive performance in large cases compared to cuSPARSE in Figure 4.19. Thus, we decided to use the packed version for half-precision CSR on H100. However, we observe some big slowdown on MI250X in Figure 4.17, so we currently choose the original version for MI250X. On Max1550, some cases give more than 5x speedup, which might be from the better sub-group size choice on the packed version. Because we can only use 1, 16, and 32 for sub-group size on Intel Max1550, it does not give a smooth selection between 1 and 16. Thus, the packed version chooses 1 for the sub-group size, which might be more suitable for the cases leading to significant speedup. We also use the packed version for Max1550.

We collect the H100 half precision CSR results comparing GINKGO and cuSPARSE in Figure 4.19. GINKGO's CSR reaches a similar peak and performs better in small cases. We only provide the performance comparison on H100 because oneMKL and hipSPARSE do not fully provide half precision CSR. We provide the supported precision set in the later Section 4.2. Full half-precision CSR means that the precision formats from A, X, Y in $Y = AX$ use half precision. cuSPARSE provides full half-precision CSR with the requirement that the scalars and computation precision must be single precision. GINKGO provides half-precision SpMV with the same precision for scalars and computation.

4.4 Summary

In this chapter, we introduce the implementation of the common sparse matrix format in GINKGO. With GINKGO's portability design in Section 3.2, we also demonstrate that the specific parameters set can be configured for different devices. GINKGO's SPMV performance is competitive with any of the other vendors' libraries but is other than the vendor libraries. GINKGO's SPMV is portable across the GPU platforms from AMD, Intel, and NVIDIA. GINKGO's SPMV supports more combinations of mixed precision than the other vendors' libraries to enable straightforward experiment usage in Table 4.2.

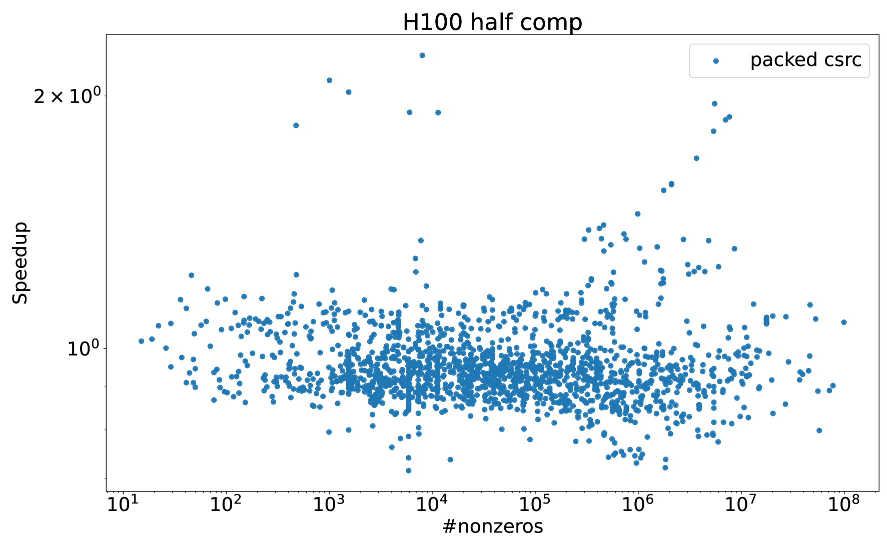


Figure 4.16: Packed version and original version performance comparison on H100.

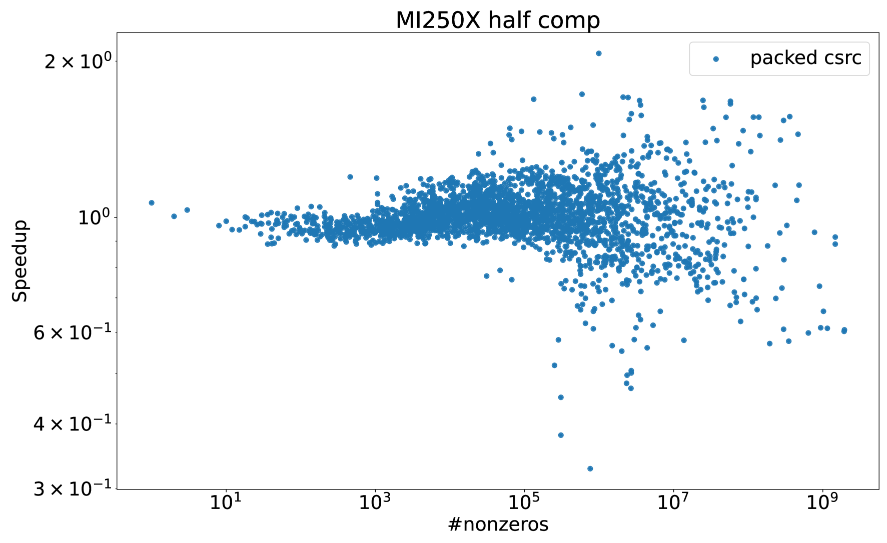


Figure 4.17: Packed version and original version performance comparison on MI250X.

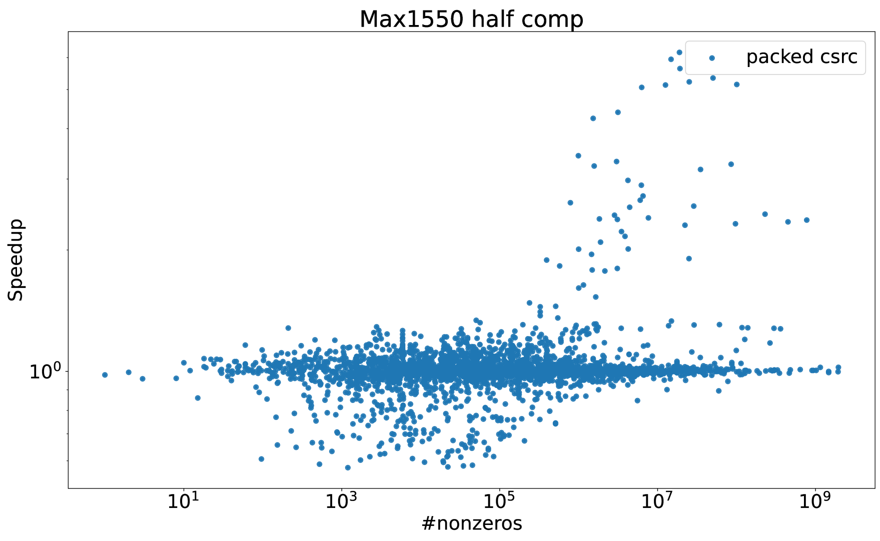


Figure 4.18: Packed version and original version performance comparison on Max1550.

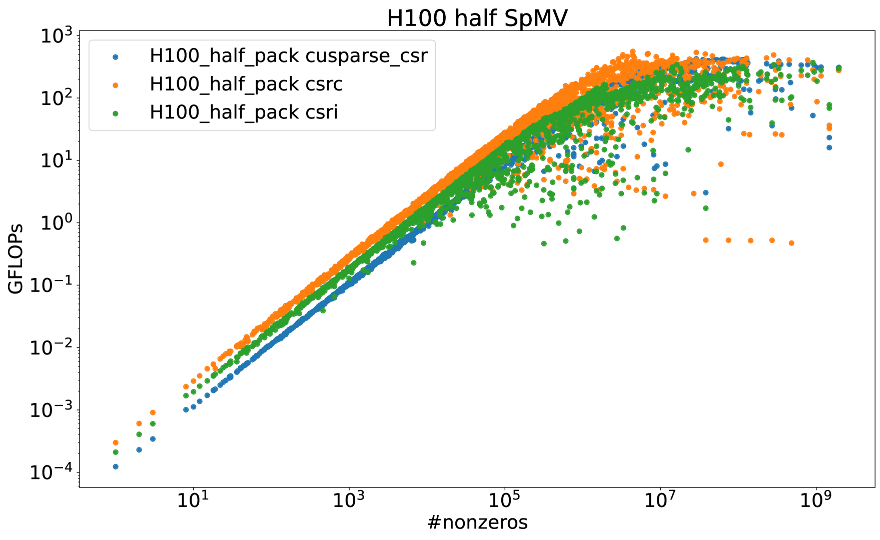


Figure 4.19: Half-Precision Csr performance comparison on H100

Algorithm 16 GINKGO'S classical CSR SPMV: packed variant with altered memory access

```
1:  Get $row$ = the row index
2:  Compute $subrow$ = the step size to next row
3:  Get $step\_size$ = the step size to the next element of value (2 * subwarp size)
4:  Get $subid$ = the local id of this subwarp.
5:  Initialize value $c = 0$
6:  for $row = row .. \#rows, row+ = subrow$ do
7:      for $idx = row\_ptr[row] .. row\_ptr[row + 1], idx+ = step\_size$ do
8:          Compute $c+ = val[idx + 2 * subid] * b[col[idx + 2 * subid]]$
9:          if idx + 2 * subid + 1 < row_ptr[row+1] then
10:             Compute $c+ = val[idx + 2 * subid + 1] * b[col[idx + 2 * subid + 1]]$
11:         end if
12:     end for
13:     Perform subwarp reduction of $c$ on the subwarp
14:     if the leading thread in subwarp then
15:         Write $c$ to the output vector
16:     end if
17: end for
```

5. High-Performance Algebraic Multigrid Design

We have introduced multigrid in Section 2.5. Because Algebraic Multigrid (AMG) is widely used in scientific applications, there are several existing AMG libraries. AmgX[Nau+15] is one of the popular AMG libraries developed by NVIDIA that provides several coarsening methods, including parallel modified independent set (PMIS) and parallel graph match (PGM), on single or multiple GPUs. The AmgX library only supports execution on NVIDIA GPUs. HYPRE[FY02] is another powerful library for distributed systems with its popular BoomerAMG[HY02] central component. HYPRE provides several interfaces for users such that users can easily use familiar formats from their domain, which avoids conversions to a generic matrix format. Currently, HYPRE supports execution on AMD, Intel, and NVIDIA GPUs. Other open-source GPU-enabled AMG implementations can be found in rocALUTION[Roc], an iterative sparse solver library developed by AMD, and MueLu[Teaa], a multigrid package inside the Trilinos[Teab] ecosystem.

Many libraries do not have a platform-portable AMG needed by many applications. NVIDIA's AmgX also has the same issue because its implementation is only available on NVIDIA hardware. Also, not all libraries have the flexibility to choose the multigrid components and precision formats for the distinct levels individually. In consequence, to enable the design of a mixed precision AMG, we develop a platform-portable high-performance AMG inside the Ginkgo library that is competitive with the vendor libraries but embraces platform portability and flexibility as central design principles.

5.1 Flexible and Platform-Portable Algebraic Multigrid

The multigrid implementation embraces the design principles of Ginkgo presented in Chapter 3: flexibility, performance, and platform portability. We use the `MultigridLevel` class to represent the essential components in the two (fine + coarse) levels of Multigrid in Figure 5.1, like the multigrid algorithm described in Algorithm 2. Each coarsening method implements the `MultigridLevel` class to generate the coarse matrix, restriction, and prolongation operations from the fine matrix. In the recursive setting, the fine matrix of the current MultigridLevel is

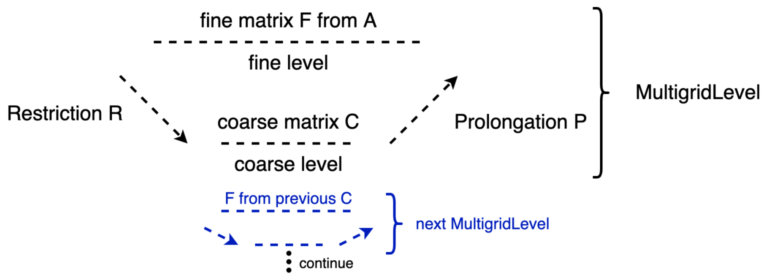


Figure 5.1: The `MultigridLevel` class with its components.

```
multigrid::build()
    .with_max_levels(10u)
    .with_min_coarse_row(64u)
    .with_pre_smoother(sm) // postsmoother use the same smoother as presmoother by
        default
    .with_mg_level(mglevel)
    .with_coarsest_solver(coarsest_solver)
```

Listing 5.1: Simple Multigrid.

the coarse matrix of the `MultigridLevel` above. As a GINKGO'S LinOpFactory design introduced in Section 3.1.2, the `MultigridLevel->generate(Fine)` generates `Prolong` × `Coarse` × `Restriction`.

The Multigrid solver provides the flexibility to set up the multigrid hierarchy from one or more `MultigridLevel`. GINKGO'S Multigrid allows a list of `MultigridLevel`, pre/post smoothers, and coarsest solvers to combine different options from users. The smoothers and the coarsest grid solvers accept LinOpFactory such that we can use the existing solver/preconditioners in GINKGO like Jacobi, Chebyshev, and direct solvers. For example, we can use this flexible configuration to generate mixed precision AMG settings by defining the different `MultigridLevel` and smoothers in different precision formats.

Listing 5.1 shows the classical multigrid settings in GINKGO. All levels will use the same smoother and the same coarsening method (MultigridLevel). This is a standard AMG V-cycle with a max multigridlevel depth of 10 (which gives a max level depth of 11), a smoother that is used for all smoothing operations, a coarsening method, and a coarse level solver(`coarsest_solver`). Note that the number of smoothing sweeps is a parameter of the smoother object `sm`.

Besides the classical multigrid settings, our flexible design allows more complex multigrid settings in Listing 5.2. The multigrid hierarchy is still the same as the classical multigrid. We choose the W-cycle for the application. Users can set multiple options in the list of presmoothers, postsmoothers, and MultigridLevels. In Listing 5.2, we set two options in presmoothers, postsmoothers, and MultigridLevel and use `with_post_uses_pre(false)` to indicate we have different smoothers between presmoothers and postsmoothers. With multiple options in the list, users can use "`with_level_selector(function)`" to set a rule for selecting the option, which can be based on the level index or the matrix properties. In Listing 5.2, we use the level index to select the option from the lists of presmoothers, postsmoothers, and MultigridLevels. The flexible design allows the setting that applies the aggressive coarsening [Stu] on the first few levels, and the rest of the levels still use the normal

```
multigrid::build()
    .with_max_levels(10u)
    .with_min_coarse_row(64u)
    .with_cycle(multigrid::cycle:w) // use w cycle instead of v cycle
    .with_pre_smoother(sm_1, sm_2)
    .with_post_smoother(sm_3, sm_4)
    .with_post_uses_pre(false) // inidicate we have different settings for post
        smoothers
    .with_mg_level(mglevel_1, mglevel_2)
    .with_level_selector(
        [](level, matrix) {
            // the first three levels (the finest level is 0) use the first option
                from smoother and mg_level
            // Otherwise, use the second option
            if (level < 3) return 0;
            else return 1;
        }
    )
    .with_coarsest_solver(iterative_solver, direct_solver)
    .with_solver_selector(
        [](level, matrix) {
            // If the size of matrix is larger than 1e5, use the first option -
                iterative solver
            // Otherwise, use the second option - direct solver
            if (matrix->get_size()[0] > 1e5) return 0;
            else return 1;
        }
    )
```

Listing 5.2: Flexible Multigrid.

coarsening. Moreover, users can also give more than one option into the coarsest solver list with the "`with_solver_selector(function)`" to select the coarsest solver based on level index or the matrix properties. In Listing 5.2, we select the iterative solver or direct solver based on the size of the coarsest matrix. We use this flexible design to construct mixed-precision AMG in Chapter 6 by setting components with different precision settings.

5.2 Performance Improvement

As smoother applications in the form of vector operations are relatively cheap, the runtime of an AMG cycle is generally dominated by the residual computations that involve sparse matrix-vector multiplications (*SpMV*s). Furthermore, we know that any memory allocation on the GPU is detrimental to performance [AMD; NVI]. To improve the performance of the AMG algorithm, we implement the following optimizations:

1. **Only compute the residual if needed.** We also implemented several optimizations regarding the handling of residual vectors. The explicit residual is not needed when using AMG as a preconditioner inside an iterative solver. In this case, the residual is only computed if the user explicitly asks for it, e.g. for monitoring convergence.

2. **Use the residual computed in a solver.** Furthermore, suppose we need the residual in the AMG solver and an internal component already computed it. In that case, this residual is accessible from outside the component to avoid recomputation. If an initial guess is zero, the residual computation is skipped, as we know the residual will be equal to the right-hand side vector.

3. **Split the termination check.** We also “split” the termination check, such that reaching an iteration limit terminates the algorithm before the residual is computed for convergence checking purposes.

4. **Avoid memory management in the execution.** We use a workspace for allocating the operator components and intermediate operations. This workspace is available over the complete lifetime of the operator. Workspace usage avoids any memory allocations or memory frees during the operator application phases.

5. **Optimize for zero initial guess**. Many iterative solvers like IR in Algorithm 7 and Chebyshev iteration in Algorithm 8 solve the residual correction equation. For a zero initial guess, we use the right-hand side as the residual vector directly because $r = b - Ax = b - 0 = b$ from a mathematical perspective. AMG uses a zero initial guess at the finest level when it is a preconditioner. Also, the rest of the levels start with a zero initial guess in Line 9 of Algorithm 2. Thus, eliminating the unnecessary residual computation from zero initial guesses is useful in AMG.

5.3 Parallel Graph Match (PGM)

In Section 2.5.2, we present two popular coarsening strategies with the corresponding popular algorithms as examples. We take the PGM algorithm Algorithm 5 to examine our AMG design and mixed precision evaluation. We detail the implementation in Algorithm 17, which involves edge-case considerations like isolating points and the controllable stopping setting.

5.4 High-Performance as the State-Of-The-Art Library

NVIDIA’s AmgX [Nau+15] is the high-performance AMG library which includes the PGM as coarsening method. It is one of the state-of-the-art AMG libraries and is widely used by CFD software like Fluent. We want to ensure our design can achieve performance competitive to NVIDIA’s AmgX but combine it with more flexibility and platform portability.

For the experimental performance evaluation, we use the MFEM library. MFEM[And+21; Mfe] is a popular open-source finite element library with support for high-order meshes and basis functions, among many other features. We pick two test cases in MFEM: L-shape and Beam in Figure 5.2 as the experiment problems. We set a constant coefficient of $c = 1$ for **L-shape** but a piecewise constant coefficient from 0.1 to 1 at the midpoint of the length of the **beam**. We set several *orders* of basis functions (passed by `-o`) and *levels* of mesh refinement (`-l`). All MFEM experiments use standard tensor-product basis functions on the Legendre-Gauss-Lobatto nodes and default choices for quadrature points based on the order of basis function. We consider a modification of MFEM’s “example 1”, solving a standard diffusion problem $-\nabla \cdot (c\nabla u) = 1$.

The experiments use MFEM’s CG solver with an AMG preconditioner either from GINKGO or NVIDIA’s AmgX. Each iteration of CG only involves one V-cycle

Algorithm 17 Parallel Graph Match Algorithm with implementation detail in [Nau+15].

```
 1: procedure PARALLEL GRAPH MATCHING($A$)
 2:     Let $Weight = \frac{A+A'}{2}$
 3:     Let $G(V, E)$ be the adjacency graph of the coefficient matrix $A$
 4:     Let $F_{sn}(i, S) := \text{argmax}_{j \in S}\{Weight(i, j)/max(Weight(i, i),$
        $Weight(j, j))\}$ determine the strongest neighbor
 5:     Let $W = V$
 6:     while $W$ is not $\emptyset$ and iteration < max_iterations do
 7:         for all $v \in W$ do
 8:             Find $v$'s strongest neighbor $w$ based on $w = sn(v)$ and check if the
                neighbor is aggregated or not.
 9:             Find $v$'s strongest neighbor in aggregated/unaggregated sets:
                the strongest neighbor in aggregated group: $F_{sn}(v, V \setminus W)$
                the strongest neighbor not in aggregated group: $F_{sn}(v, W)$
10:             if All neighbors are aggregated then
11:                 Merge $v$ into the aggregated group of the strongest neighbor
12:                 Remove $v$ from the next outer iteration, so that $W = W \setminus \{v\}$
13:             else if Has a non-aggregated strongest neighbor then
14:                 Store $sn(v) := F_{sn}(v, W)$ for next operation
15:             else                                                    ▷ no neighbor
16:                 Assign the strongest neighbor as itself: $sn(v) := v$ such that it will
                    be formed as an isolated aggregation in the next operation
17:             end if
18:         end for
19:         for all $v \in W$ do
20:             if $sn(sn(v)) = v$ then
21:                 Form aggregation, which takes the smaller of the row and column
                    indices as the identifier
22:                 Remove them from the next outer iteration so that $W = W \setminus$
                    $\{v, sn(v)\}$
23:             end if
24:         end for
25:     end while
26:     if $W$ is not $\emptyset$ then
27:         Assign the rest of the unaggregated nodes to the aggregated group of its
            strongest neighbor or form an isolated group when it is not connected to others
28:     end if
29:     Assign the aggregation index to each aggregation group
30:     Build the prolongation $P$, which $p_{vi} = 1$ when $v$ is in $i$ aggregated group
31:     Get the restriction $R = P^T$ and Coarse matrix $C = R \times A \times P$
32: end procedure
```

Figure 5.2: Meshes used for MFEM diffusion experiments. Left: L-shape mesh with 7 levels of uniform refinement (49,152 elements); Right: Beam mesh with 3 levels of uniform refinement (4,096 elements).

	problem	size	nonzero elements
	beam (-o2 -l3)	37,281	21,67,425
	beam (-o3 -l3)*	120,625	14,070,001
	beam (-o4 -l3)	279,873	57,251,713
matrices in MFEM integration test	beam (-o3 -l4)	924,385	111,573,601
	L-shape (-o3 -l7)*	443,905	11,066,881
	L-shape (-o3 -l8)	1,772,545	44,252,161
	L-shape (-o4 -l7)	788,481	28,323,841
	L-shape (-o4 -l8)	3,149,825	113,270,785
	2cubes_sphere	101,492	1,647,264
	thermal2	1,228,045	8,580,313
SuiteSparse matrices for platform-independent tests	cage14	1,505,785	27,130,349
	cage13	445,315	7,479,343
	offshore	259,789	4,242,673
	tmt_sym	726,713	5,080,961

Table 5.1: Matrix characteristics for the selected MFEM discretizations and Suite Sparse Matrix Collection matrices. The MFEM matrices marked with (*) were exported from MFEM and also used in the platform-independent tests.

application of AMG. To generate the same preconditioner, we use the same setting and algorithm for Ginkgo and NVIDIA's AmgX. The AMG uses the following settings:

- coarsening: PGM with deterministic aggregation of size 2
- hierarchy limitation: 10 maximum number of levels (which is a parameter value in Ginkgo, but AmgX uses 11 to have the same hierarchy limitation) or 64 for minimum matrix size. If the generation reaches one of the limits, it will stop generations.
- pre-/post-smoothing: 1 iteration weighted Jacobi smoother (weight = 0.9).
- coarsest solver: 4 iteration weighted Jacobi smoother (weight = 0.9).

we consider several *orders* of basis functions (`-o`) and *levels* of mesh refinement (`-l`), which define the problem names in Table 5.1.

In Table 5.2, we list the iteration counts and runtime performance for different CG/preconditioner configurations on the NVIDIA V100 GPU. We first focus on four discretizations for the **Beam** geometry on the top of Table 5.2. The iteration counts of the AMG-preconditioned CG solver are generally similar. Because Ginkgo does not use the same SpMV kernel as NVIDIA, different rounding errors may

geometry	problem	NVIDIA AmgX runtime [ms]	NVIDIA AmgX #iter	GINKGO AMG runtime [ms]	GINKGO AMG #iter
Beam	-o 2 -l 3	20.71	15	20.27	15
	-o 3 -l 3	52.94	20	39.93	21
	-o 4 -l 3	155.47	26	128.69	27
	-o 3 -l 4	329.68	29	294.68	29
L-shape	-o 3 -l 7	242.27	93	178.02	93
	-o 3 -l 8	1211.38	180	1033.96	173
	-o 4 -l 8	3452.91	251	3044.24	236
	-o 4 -l 7	551.99	129	407.27	122

Table 5.2: MFEM **Beam** (top) and **L-shape** (bottom) examples using MFEM's AMG-preconditioned CG solver on NVIDIA V100 GPU. Both AMGs are executed in IEEE double precision. The data is from our previous work [TBA23a]

lead to slightly different convergences. CG preconditioned with NVIDIA's AmgX preconditioner converges one iteration sooner in two cases. However, it is more expensive per iteration than GINKGO'S AMG preconditioner. In the end, GINKGO'S AMG is approximately 20-40% faster than NVIDIA's AmgX for the three larger problems in terms of total solving time. Compared to the **Beam** geometry, the **L-shape** geometry is numerically more challenging due to its re-entrant corner. We use the same experiment settings and report the results in the bottom part of Table 5.2. Here, the trend of the AmgX-preconditioned CG requiring fewer iterations is reversed, as in this case, GINKGO's AMG enables faster convergence. Combined with the faster preconditioner application per iteration, GINKGO's AMG gets an attractive runtime improvement over AmgXfor all discretizations of the **L-shape** geometry. The total speedup with the **L-shape** geometry is approximately 20-40% from GINKGO'S AMG. Overall, the Ginkgo AMG accelerates the solution process by 20%-40% compared to NVIDIA's AmgX.

We also evaluate GINKGO'S AMG and NVIDIA's AmgX as stand-alone solvers. The hierarchy of AMG is still the same setting as the above preconditioner. We also use a relative residual stopping criterion of 10^{-9} and allow for at most 100 AMG iterations. We export the MFEM matrix and the corresponding right-hand side from the (-o3 -l3) configuration of the Beam geometry and the (-o3 -l7) configuration of the L-shape geometry. Additionally, we use selected matrices from SuiteSparse[DH11]. In Table 5.3, We also compare against the popular academic AMG library HYPRE[FY02] in Table 5.3. However, HYPRE has its own coarsening method such as PMIS, which is introduced in Section 2.5, and PMIS is different from PGM. We add both coarsening strategies to the comparison to demonstrate that they can result in different convergence characteristics.

We summarize the performance of the standalone AMG solver on V100 in Table 5.3. Except for 2cubes_sphere problem, GINKGO'S AMG is competitive or outperforms NVIDIA's AmgX. HYPRE PMIS shows a different convergence behavior than PGM, indicating that different applications may need different coarsening algorithms for good performance and convergence. HYPRE here is only as a reference for different algorithms, so discussing the difference between PMIS and PGM is out of the scope of this dissertation.

	NVIDIA AmgX			GINKGO's AMG			HYPRE AMG		
problem	[ms]	#it	res.norm	[ms]	#it	res.norm	[ms]	#it	res.norm
beam(o313)	199.44	87	9.03e-10	152.87	84	9.75e-10	82.394	30	9.68e-10
L-shp.(o317)	251.01	100	7.43e-04	236.43	100	6.89e-04	67.892	33	9.99e-10
2cubes.	130.14	88	7.94e-10	160.63	91	8.88e-10	187.48	100	1.33e-8
thermal2	284.73	100	1062.42	304.79	100	1062.74	327.42	100	5.7206
cage14	79.30	15	4.28e-10	84.01	14	6.73e-10	119.67	*86	8.46e-10
cage13	40.99	17	7.29e-10	42.61	18	5.37e-10	44.87	*87	8.68e-10
offshore	180.92	100	1.76e33	172.48	100	1.95e33	236.34	100	inf
tmt_sym	211.65	100	858.151	197.63	100	858.84	265.46	100	1.45e6

Table 5.3: Comparison of performance, convergence, and accuracy of different AMG solvers on the NVIDIA V100 GPU. For HYPRE, we mark * on iteration when HYPRE does not generate any level for the problem. All AMGs are executed in IEEE double precision. The data is from our previous work [TBA23a].

	GINKGO's AMG on AMD MI100			GINKGO's AMG on Intel P630		
problem	[ms]	#it	res.norm	[ms]	#it	res.norm
beam (o313)	235.23	84	9.80e-10	3535.40	85	8.53e-10
L-shape (o317)	272.28	100	6.88e-04	5756.48	100	6.89e-04
2cubes.	130.63	91	8.88e-10	1806.15	91	8.88e-10
thermal2	287.12	100	1062.74	6715.82	100	1062.7
cage14	105.50	14	6.75e-10	3449.86	14	6.77e-10
cage13	62.27	18	5.37e-10	1031.46	18	5.37e-10
offshore	207.69	100	1.95e33	3155.41	100	1.95e33
tmt_sym	208.72	100	858.843	4043.05	100	857.767

Table 5.4: Comparison of performance, convergence, and accuracy of GINKGO's AMG solver on the AMD MI100 GPU and Intel P630 GPU. Both AMGs are executed in IEEE double precision. Intel P630 GPU is an integrated GPU, so we do not expect the performance of Intel P630 good. The data is from our previous work [TBA23a]

Other than NVIDIA's AmgX, GINKGO's AMG is portable to other accelerators. We collect the same experiment results on AMD MI100 and Intel Gen9 P630 GPU in Table 5.4.

5.5 Summary

We designed a portable high-performance AMG which is competitive with the existing state-of-the-art NVIDIA's AmgX. We achieve this by using the hardware-native language to implement the hardware-specific kernels inside the generic AMG framework. Compared to vendor libraries like NVIDIA's AmgX, GINKGO's AMG does not only come with platform portability, but also with more flexibility in terms of choosing the multigrid components like smoothers and the precision format in the distinct levels, resulting in the mixed precision AMG that we will discuss in Chapter 6.

6. Mixed Precision Algebraic Multigrid

In this chapter, we combine the building blocks from the previous Chapter 5 to design a high performance mixed-precision AMG. All the building blocks are designed as platform-portable components such that the resulting mixed-precision AMG can run on GPUs from different vendors and general-purpose CPUs. The mixed precision SpMV design allowing for input data with different precision formats allows to implement the mixed precision AMG without explicitly forming additional copies of the matrices. We start mixed-precision AMG from a simple idea, which sets each level with different precision in Section 6.3. It works well with double precision AMG mixed with some single precision operations. However, it encounters numerical challenges when using half precision for the lower multigrid levels. To resolve these issues, we designed a new mixed-precision scheme in Section 6.4, which allows different precision within a single level. Another idea to resolve these issues is adding a non-IEEE standard bfloat16 precision format, which uses 16 bit as the half precision but has the same range as the single precision, in Section 6.5. We do not only try the V-cycle AMG-preconditioned CG experiments, but also standalone mixed-precision AMG in Section 6.6 and W-cycle AMG-preconditioned CG in Section 6.7. We demonstrate the mixed-precision AMG can improve the performance and reach the same accuracy as the double precision AMG on high-performance GPUs.

Listing 6.1 shows the setup of mixed precision AMG with two `MultigridLevels` and smoothers using different precisions. Similar to Listing 5.2, we can set different components for the multigrid hierarchy. The difference between the components is the precision format, we select the preicison format for smoothers and MultigridLevel for each level individually. Listing 6.1 shows how to generate an AMG configuration that uses DP for the first level and SP for all subsequent levels. We give the double precision smoother and MultigridLevel, and single precision corresponding version into the option list. We provide the function to select the first option (double precision components) at the first (finest) level but the second option (single precision components) at the rest of the levels. Assuming we always generate more than 1 level, we put the single precision solver as the coarsest solver.

```
multigrid::build()
    .with_max_levels(10u) // equal to NVIDIA/AMGX 11 max levels
    .with_min_coarse_row(64u)
    .with_pre_smoother(sm, sm_f)
    .with_mg_level(pgm, pgm_f)
    .with_level_selector(
        [](const size_type level, const LinOp*) -> size_type {
            // Only the first level is generated by MultigridLevel(double).
            // The subsequent levels are generated by MultigridLevel(float)
            return level >= 1 ? 1 : 0;
        })
    .with_coarest_solver(coarest_solver_f)
```

Listing 6.1: Mixed precision Multigrid.

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

Table 6.1: Matrix characteristics for the selected problems.

In Chapter 5, we have demonstrated that GINKGO's AMG is competitive to NVIDIA's AmgX. In the following experiments, we will focus on the enhanced flexibility and mixed precision options of GINKGO's AMG. We run the following experiments on three GPUs - NVIDIA H100 (PCIE), AMD MI250X (1GCD), Intel Max1550 (1 tile) with their characteristics in Table 2.3 and corresponding environments in Section 2.7. We run the experiments with 2 warm-up iterations and collect the median value from the 5 repetition iterations.

6.1 Experimental Settings

As the experiments in Chapter 5, we use GINKGO as the main application to test the performance across several platforms. We take the same matrix set as the previous chapter, that is two configurations of MFEM cases and matrices from the SuiteSparse Matrix Collection [DH11]. Table 6.1 shows the characteristics of the test matrices including the range of values.

The maximum number of multigrid levels is 11. If the coarse matrix needs less than 64 rows, it will stop the hierarchy generation. For the AMG-preconditioned conjugate gradient (CG) solver, we set the stopping criterion as implicit relative residual norm reduction of 10^{-12} or a maximum of 800 iterations. We have two kinds of smoothers for pre-/post-smoothing. One is the weighted scalar Jacobi with a weight of 0.9 in Algorithm 9. The other is ℓ_1-Jacobi-Chebyshev which uses the ℓ_1-Jacobi as the inner solver of Chebyshev iteration in Algorithm 8.

We apply 1 iteration for pre-/post-smoothing but 4 iterations for the coarsest solver with the weighted Jacobi smoothers. We apply 2 iterations for pre-/post-smoothing and the coarsest solver with ℓ_1-Jacobi-Chebyshev.

6.2 SpMV Performance as a Proxy for AMG

For an idea of the potential performance benefits of mixed precision AMG on our target architectures, we analyze the performance of the sparse matrix-vector operation (SpMV) used in residual computations and smoothers. We expect the residual computations on each level to be the major limiting factor of performance; in comparison, the solution phase of our smoothers, as well as the restriction and prolongation operations, are simpler. The SpMV is a memory-bound operation, with performance tied to the amount of memory traffic required rather than the number of floating point operations performed. The memory involved in a SpMV operation of an $n \times n$ CSR matrix with nnz non-zero elements can be written as:

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

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

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

For example, the speedup of single-precision SpMV over double-precision SpMV would be 1.5x, and the speedup of half-precision SpMV over double-precision SpMV would be 2x in theory. We extract the original matrices and the coarse matrices from levels 0-10 in the 11-levels AMG hierarchy from the problems in Table 6.1. We follow the same experimental settings as in Section 6.2 to collect the SpMV performance. We run two warmup iterations and use the median value of 10 iterations repetitions. Figure 6.1 shows the speedup from single and half-precision on H100. The single and half-precision SpMV show similar performance and do not have a big speedup against double precision SpMV on the small matrices at the coarse levels. In the cage13, cage14, offshore, and l-shape problems, the single precision cases can reach around 1.5x speedup, and the half precision cases can reach around 2.0x speedup at the fine levels. In the thermal2 and tmt_sym problems, half-precision SpMV can reach close to 3.0x. The higher speedup in practice than expected may be caused by the cache fitting or the half subwarp size performing better on these problems. We also collect performance data on MI250X (1GCD) in Figure 6.2. Thermal2 and tmt_sym problems show a noticeable difference between single precision and half-precision at fine levels, which may be from the same reasons as on H100. The single precision SpMV shows better performance than double precision SpMV. The difference between single and half-precision on H100 is larger than on MI250X because HIP does not support 16-bit shuffle natively. Figure 6.3 shows the performance difference from the three precisions on Max1550 (1tile). Only a few cases like 2cubes_sphere, cage13, and cage14 show speedups.

We can compute the storage approximation for an entire AMG cycle based on the number of SpMVs on each level. In our implementation, the coarsening usually aggregates two nodes together, while the exact compression ratio depends on the sparsity pattern. Figure 6.4 reports the relative nonzero count of the matrices in the

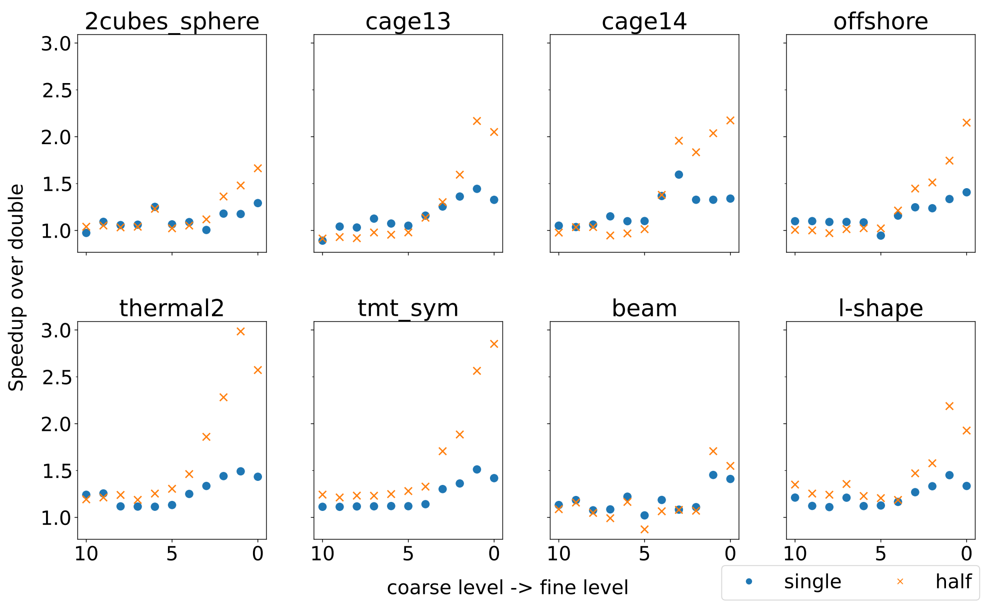


Figure 6.1: H100 speedup of single and half-precision SpMVs compared to double precision for the matrices produced on each level of AMG. The half-precision SpMV uses the "packed half" version

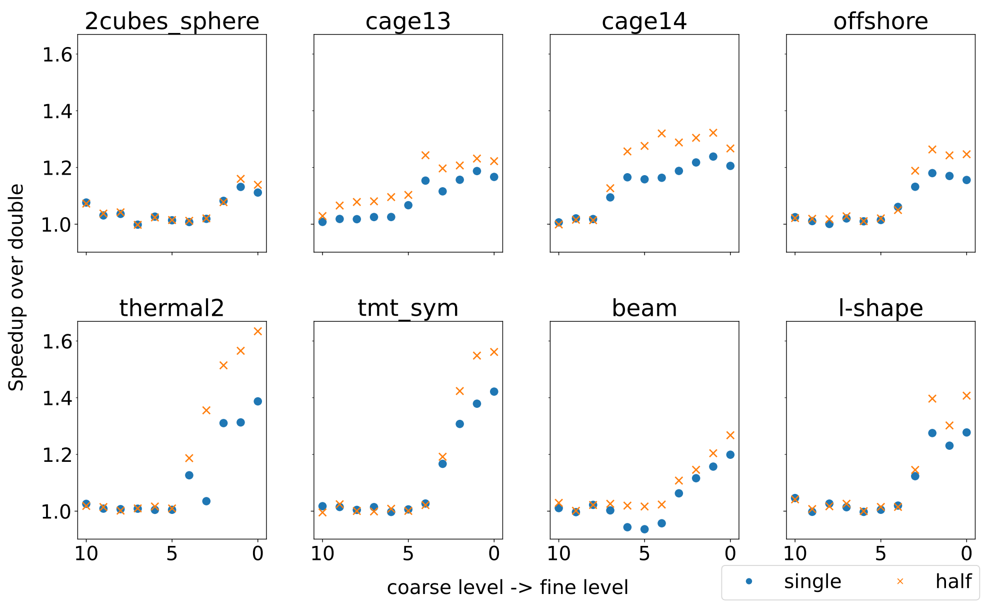


Figure 6.2: MI250X speedup of single and half-precision SpMVs compared to double precision for the matrices produced on each level of AMG.

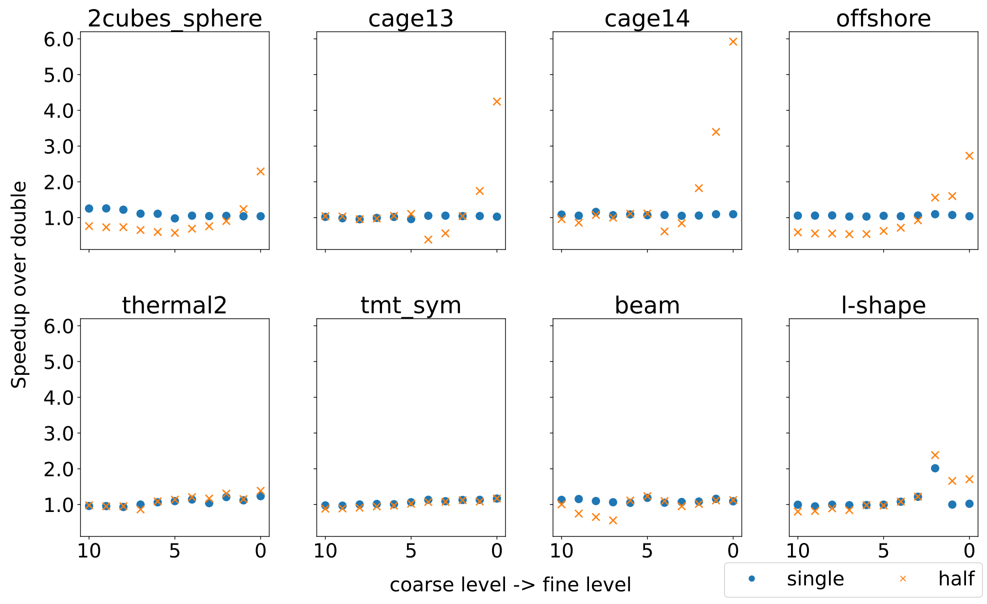


Figure 6.3: Max1550 speedup of single and half-precision SpMVs compared to double precision for the matrices produced on each level of AMG. The half-precision SpMV uses the “packed half” version

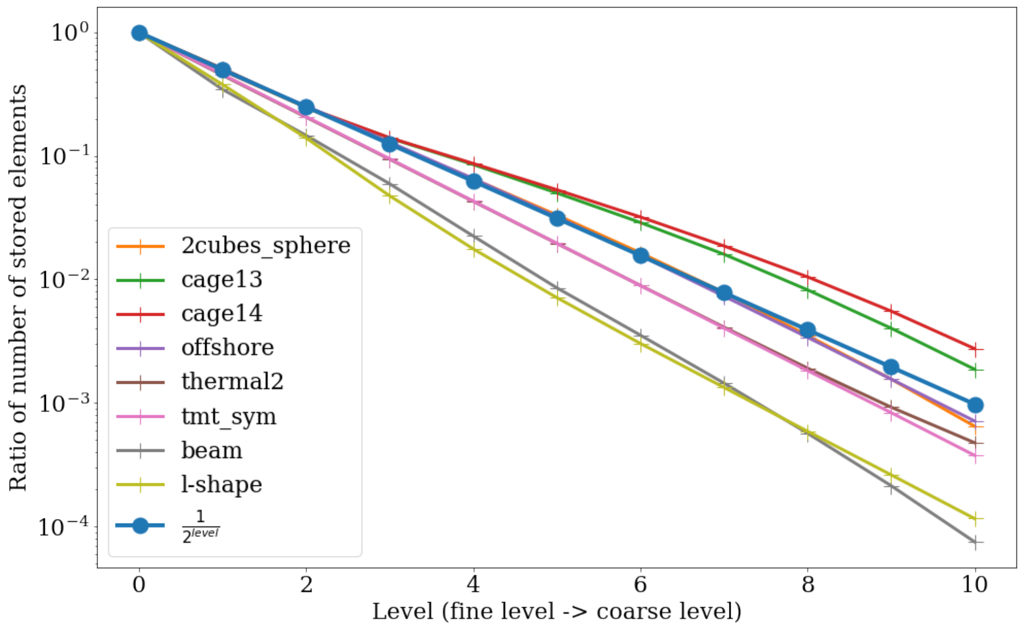


Figure 6.4: The ratio of the number of stored elements in each level of a multigrid hierarchy to that of the finest level (level 0), using Parallel Graph Match [Nau+15] with size 2 for aggregation.

distinct AMG levels compared to a compression ratio of 2. Accumulating over an AMG hierarchy of $N + 1$ levels, we have

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

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

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

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

6.3 Uniform Level Configurations

Algorithm 18 V-cycle multigrid method with color notation for precision. We use blue, red, and brown colors, respectively, to indicate the precision for: matrices (A), working vectors on the fine level (r), and working vectors on the next coarsest level (g, e). The presmoothers and postsmoothers also use the working precision for computation and matrix precision for the system matrix.

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

We implement the mixed precision AMG based on a simple idea. We do not need to solve the coarse level very accurately because the coarse matrix is not the original matrix. We try to use lower precision alternatively to achieve higher performance at coarse levels. Uniform level configurations imply that the level uses the same precision for the matrix A, residual r, initial guess x, and the right-hand side b in Algorithm 18. The precision changes are on the fly with prolongation and restriction because the explicit copy contains overhead and reduces the performance. In Algorithm 18, the uniform precision configuration prepares vector g and e (brown color) for the next coarse level precision. We denote DP - double precision, SP - single precision, and HP - half precision as in Table 4.1. For the level settings, we describe the settings from the finest level. If there is any following precision, it changes the precision after one level. We have the following settings in this section:

- *DP*: all levels use double precision.

	Ginkgo's AMG (DP)			Ginkgo's AMG (DP-SP)			Ginkgo's AMG (DP-SP-HP)			Ginkgo's AMG (DP-HP)		
problem	res. norm	#iter	time[ms]	res. norm	#iter	time[ms]	res. norm	#iter	time[ms]	res. norm	#iter	time[ms]
2cubes	6.56e-09	20	14.62	6.56e-09	20	12.91	NaN	800	493.44	NaN	800	485.06
cage13	3.68e-10	11	15.49	3.68e-10	11	14.09	4.24e-10	11	14.18	6.15e-10	15	17.79
cage14	3.41e-10	10	32.98	3.41e-10	10	29.80	3.73e-10	10	29.04	7.02e-10	13	35.15
offshore	1203.64	800	770.56	1533.85	800	688.64	NaN	800	706.05	NaN	800	671.65
thermal2	2.18e-06	349	489.08	2.38e-06	425	536.09	NaN	800	952.22	NaN	800	919.08
tmt_sym	6.95e-05	359	371.70	7.41e-05	401	379.21	NaN	800	703.40	NaN	800	684.14
beam(o3l3)	3.05e-15	44	43.65	3.05e-15	44	40.89	5.53e-15	86	80.72	7.27e-15	127	117.62
l-shape(o3l7)	4.64e-14	160	195.31	4.78e-14	171	191.62	3.74e-11	800	869.77	2.19e-07	800	868.83

Table 6.2: Performance of CG preconditioned with an AMG V-cycle, scalar Jacobi smoother with uniform level precision configurations, on H100. The "packed half" SpMV implementation was used.

- $DP - SP$: the first level uses double precision and the rest of the levels use single precision
- $DP - SP - HP$: the first level uses double precision, the second level uses single precision, and the rest of the levels use half precision
- $DP - HP$: the first level uses double precision and the rest of the levels use half precision.

From Table 6.2, we see that mixed precision AMG with single precision performs well because we get performance improvement without facing any convergence issue in these cases. When adding half-precision, mixed-precision AMG faces convergence challenges in many cases. We observe the following reasons for these challenges and provide a solution.

- The matrix values of 2cubes_sphere and offshore are originally out of the range of half-precision such that we get $Inf * 0 = NaN$ when applying the matrix in half precision with a zero vector.
- The Jacobi smoother in thermal2 and tmt_sym is out of the range of half-precision. Some diagonal values are within the range of half precision, but the inverted values are not.
- The residual of L-shape is too small for coarse levels leading underflow issue of half precision.
  When the residual changes the precision after restriction, the residual becomes zero in half precision due to underflow. For the coarse level, this means that the right-hand side is zero such that the linear system has a trivial solution - all zeros. Thus, the coarse levels only prolongate the zero update to fine levels, which does not contribute to the overall correction. L-shape does not face the NaN issue like the other cases, but the convergence with half precision is slower than the other configurations. The underflow issues may appear after a few iterations, not from the beginning.

We use three different strategies to overcome the numeric challenges. First, we apply row/column scaling as Algorithm 2.5 of [HPZ19], detailed in Algorithm 19, to the 2cubes_sphere and offshore matrices to ensure all values are in the range of half precision range. We would like to compare the convergence of higher precision and mixed precision directly, so the scaling on these matrices is applied for all precision configurations, not just those involving half. In PGM, we add a scaling of 1/2 to the restriction matrix to avoid the values being out of the half precision. The (DP-SP-HP) configuration can solve the 2cubes_sphere after scaling the matrix. Secondly, we decouple the precision setting in the level in Section 6.4 to use higher precision in the vectors to avoid the underflow/overflow issues. Third, because the challenges are mainly from the small range of IEEE half precision, we use the bfloat16 precision format alternatively in Section 6.5, which has the same range as single precision in Table 2.2.

Algorithm 19 Symmetry-preserving row and column equilibration (one iteration of Algorithm 2.5 in [HPZ19]).

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

6.4 Non-Uniform Level Configurations

We split the precision requirement between working vectors and matrix storage to use the higher precision for working vectors but still lower precision in matrices. By using higher precision in the working vectors, the residual computation is in higher precision to combat roundoff errors. Also, the higher precision in residual vectors avoids underflow issues, which was leading to zero vectors in half precision. Moreover, the smoother precision follows the working vector precision, so we do not encounter the overflow issue when inverting diagonal values. We still retain most of the benefit from half precision in SpMV because the matrix memory footprint is the primary performance concern in SpMV. However, decoupling precision incurs some extra cost and memory usage in the generation step once because it passes the matrix with the highest precision to the next coarse level generation and the level generation needs to keep both of them for different usages. However, it improves the performance with fewer convergence issues for applications that are used several times.

Non-uniform level configurations are configurations where the matrix and the vectors do not use the same precision format. The mixed-precision SpMV will use the highest precision format among the input, output, and matrix precisions for the arithmetic operation as we mentioned in Section 4.2. The mixed precision application will change the precision on the fly without any explicit conversion and write the desired precision to the output memory directly. In these configurations, the working vector precision always uses more bits than the matrix precision, so the working vector precision always indicates the precision for arithmetic operations. The notation here is similar to the configuration in Section 6.3. We consider the following non-uniform configurations:

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

We have 6 possible combinations from these two lists. In the following discussion, we use the notation *(Working Precision, Matrix Precision)* to represent the precision selection in the configuration. To use the same notation style, we map the uniform level configuration to repeat settings in both entries. For example, (DP-HP) in the uniform level configuration will be represented as (DP-HP, DP-HP) in the following discussion.

We notice that the offshore test matrix fails in all configurations in Table 6.2. In this section, we set up another more complicated smoother - ℓ_1-Jacobi-Chebyshev to solve the offshore system. The ℓ_1-Jacobi-Chebyshev smoother uses the symmetric positive definite (SPD) property of the matrix in Section 2.6. We omit the cage13 and cage14 test cases in the ℓ_1-Jacobi-Chebyshev experiments because they are not SPD matrices. We also omit the offshore test case in the weighted Jacobi smoother because it does not converge in the complete double precision setting. The ℓ_1-Jacobi-Chebyshev can improve the convergence of the offshore problem within the 800 iterations limit.

In the following section, we present the speedup information and the factors of performance and convergence in one figure. First, we show the speedup in the bar plot from mixed precision against the double precision. If the case name contains "*", the problem is pre-scaled to fit the half-precision range. Suppose a certain configuration does not give any speedup against a complete double-precision configuration. In that case, it will not have a bar for that configuration. We embed additional information into the speedup figures with the notation in Table 6.3. Suppose there is a bar for one configuration with $\circ$ label. In that case, the configuration is still faster than the complete double configuration within more iterations. That is, the mixed-precision improvement can cover the additional iterations. Suppose there is no bar for one configuration with $\circ$. In that case, the configuration is slower than the complete double configuration because the configuration needs more iterations to converge.

On H100, we collect the performance data of AMG preconditioned CG with Jacobi smoothers in Figure 6.5 and with ℓ_1-Jacobi-Chebyshev smoothers in Figure 6.6. Except for (DP-HP, DP-HP), all mixed-precision configurations performs better than double-precision configuration with 2cubes_shpere problem. Although (DP-HP, DP-HP) performs better per iteration with 2cubes_shpere problem, it needs more iterations to converge such that the performance is slower than the double-precision configuration. It is the same reason why some configurations miss a speedup bar

label	meaning
NaN ($\times$)	the result residual norm of the configuration is NaN
More iter ($\circ$)	the configuration requires more iterations than the complete double precision configuration
No speedup per iter ($\triangle$)	the time per iteration of the configuration is not faster than the complete double precision configuration
Not converged (∇)	the configuration does not converge

Table 6.3: The label meaning in the speedup figures.

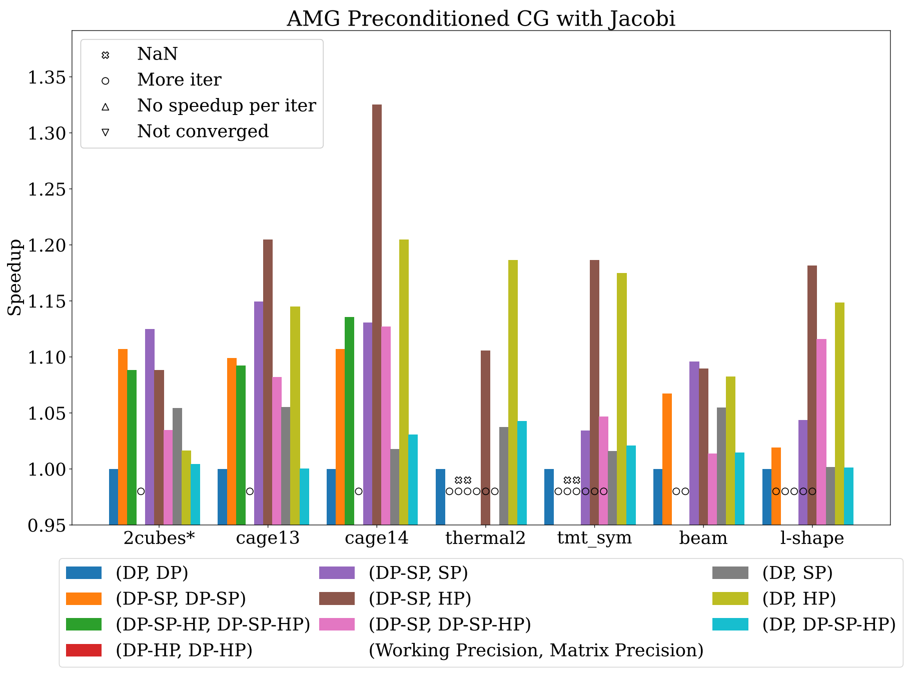


Figure 6.5: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving. The detailed meanings of labels are available in Table 6.3.

but with $\circ$(More Iter) mark only, such as (DP-HP, DP-HP) with cage13 and cage14 problems. With thermal2 and tmt_sym problems, the uniform configuration with half precision shows the NaN mark($\times$), which we discussed in the Section 6.3. Except for the (DP-SP, DP-SP) configuration, all (DP-SP, *) configurations with the tmt_sym problem need more iterations than double precision configuration. However, the speedup per iteration of mixed-precision can cover the additional iterations. The tmt_sym and thermal2 problems require double precision for working precision to use the same number of iterations for convergence. The l-shape problem needs more iterations with (DP-SP, SP) and (DP-SP, HP) but not with (DP-SP, DP-SP-HP). We can get up to 1.15x speedup with single precision and up to 1.35x speedup with half precision. The non-uniform level configuration performs better in terms of performance and convergence than the uniform level configuration.

We collect performance data with ℓ_1-Jacobi-Chebyshev smoothers in Figure 6.6 on H100. We can solve the offshore problem with ℓ_1-Jacobi-Chebyshev smoothers. For

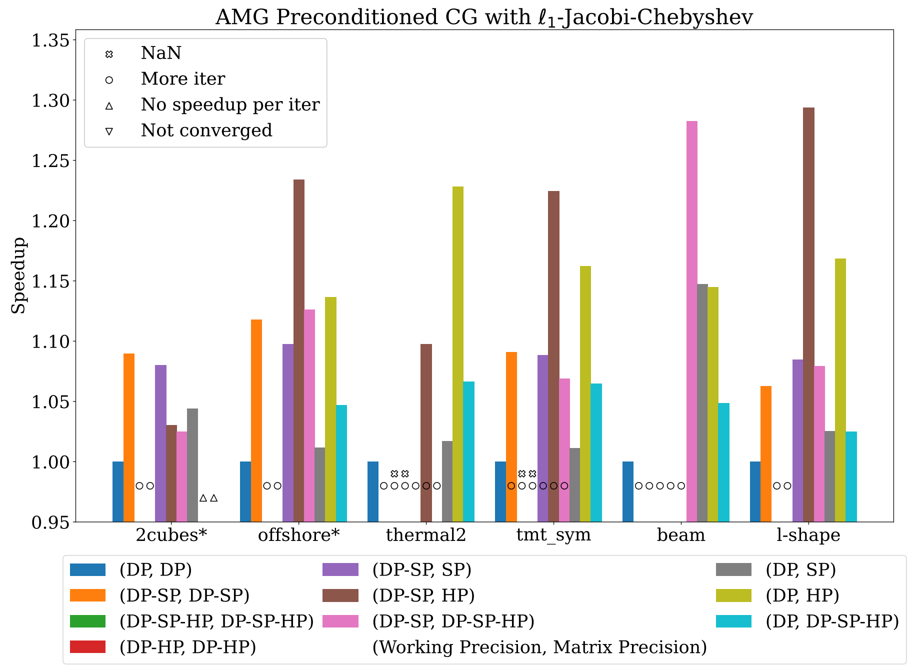


Figure 6.6: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving. The detailed meanings of labels are available in Table 6.3.

the offshore problems, the uniform configuration with half precision needs more iterations to reach the required stopping criterion although it still uses less time per iteration. For the thermal2 and tmt_sym problems, we have a similar story in terms of performance to the Jacobi smoothers. The beam and l-shape problems show fewer speedup bars because the speedup per iteration is insufficient to cover the additional iterations. Mixed precision configurations show up to 1.3x speedup compared to the double precision configurations.

We also give the iteration counts in Figures 6.5 and 6.6 on H100. For both smoother options, the (DP-HP, DP-HP) configuration always needs the most number of iterations because it is the earliest one to change the working and matrix precision to half. Although (DP-SP, HP) and (DP, HP) change the matrix precision at the finest level, they do not affect the convergence. The thermal2 and tmt_sym problems using (DP, *) for the working precision give the same convergence no matter if the matrix is in lower precision. From the above observation, the working precision is more important for the convergence than the matrix precision. (DP-SP, DP-SP-HP) on the beam problem with ℓ_1-Jacobi-Chebyshev smoothers takes fewer iterations than double precision, in which case rounding error has some positive effect and makes a more impactful update at the coarse levels with lower precision.

Figure 6.9 shows the performance overview for AMG preconditioned CG with Jacobi smoothers and Figure 6.10 shows the overview for AMG preconditioned CG with ℓ_1-Jacobi-Chebyshev smoothers. For 2cubes_sphere, half precision renders only small performance advantages over single precision. The (DP-SP, HP) configuration shows that the highest speedup is around 1.12x with the cage14 problem. Because CSR with pack/no-pack versions or different subwarp sizes (in Section 4.3.2) can

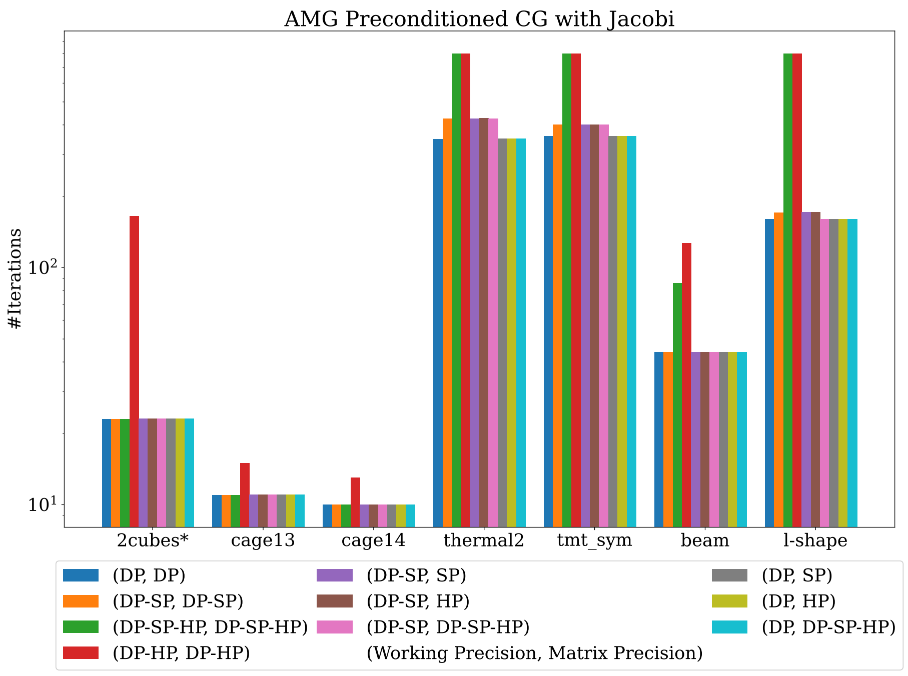


Figure 6.7: Total iterations for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are shown for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

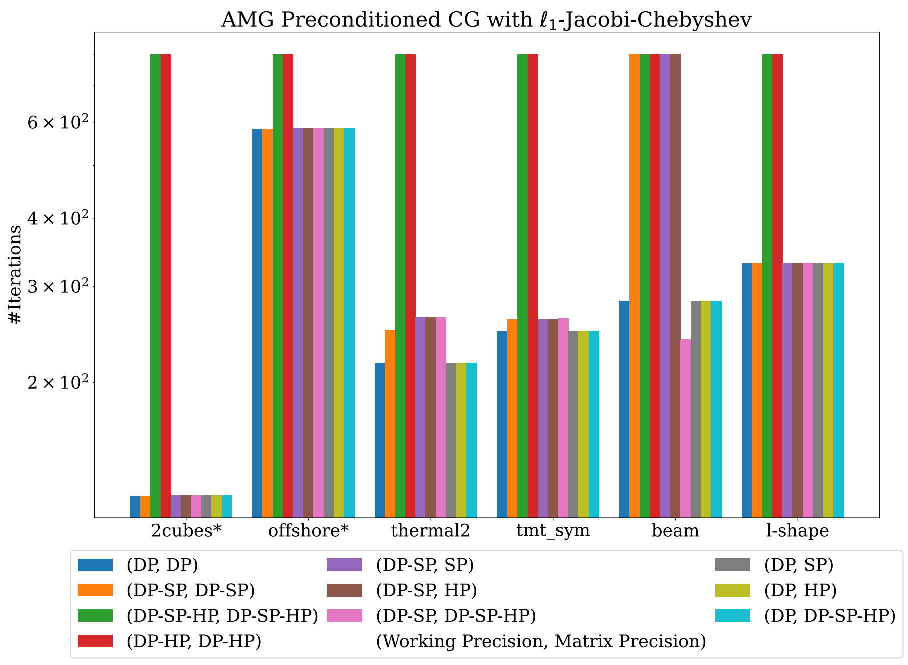


Figure 6.8: Total iterations for CG with AMG V-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration. Results are shown for H100 with the packed half SpMV variant. * denotes the matrix was scaled prior to solving.

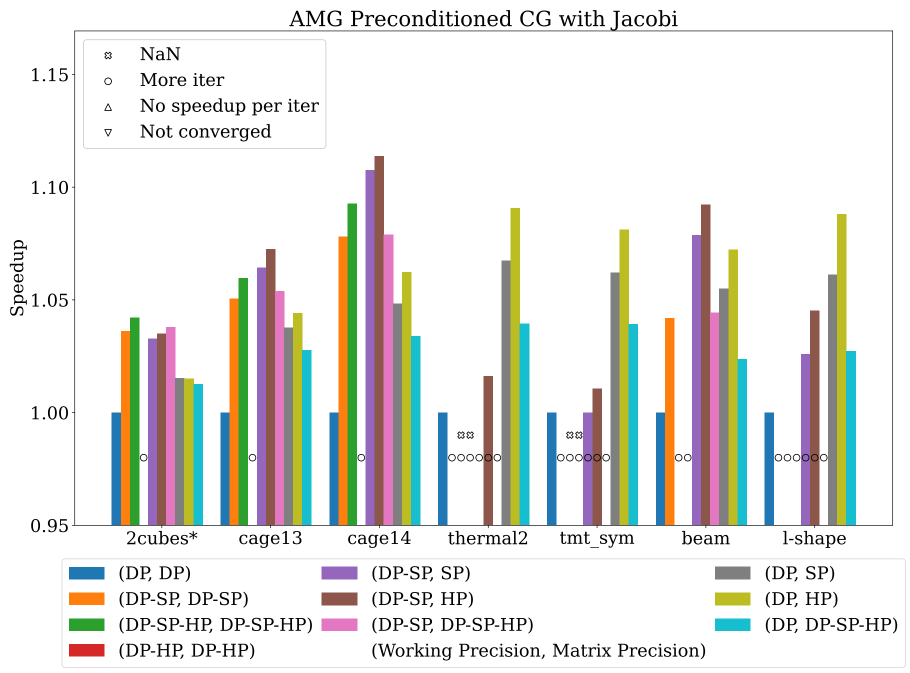


Figure 6.9: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for one GCD of MI250X. * denotes the matrix was scaled prior to solving. The detailed meanings of labels are available in Table 6.3.

introduce different accumulations order and lead to different summations in the end. (DP-SP, DP-SP-HP) needs more iterations in l-shape case on MI250X, but not on H100. With mixed precision, we can reach up to 1.125x speedup on MI250X with Jacobi smoother settings. In Figure 6.10, (DP-SP, SP) for the beam problem shows a quite significant speedup against the other problems because the single precision matrix may fit in the GPU cache well. Except for that, the other mixed precision cases can also show up to 1.12x speedup.

We show the performance data on Intel Max1550 in Figure 6.11 for Jacobi smoothers settings and Figure 6.12 for ℓ_1-Jacobi-Chebyshev smoothers. Unlike H100 and MI250X, we observe speedup mainly for the cage14 problems. Only a few mixed precision configurations show speedup for the other problems. It might be related to the larger overhead of kernel execution or the fewer options for sub-group size than the other hardware such that the speedup of the mixed-precision usage does not play a major role.

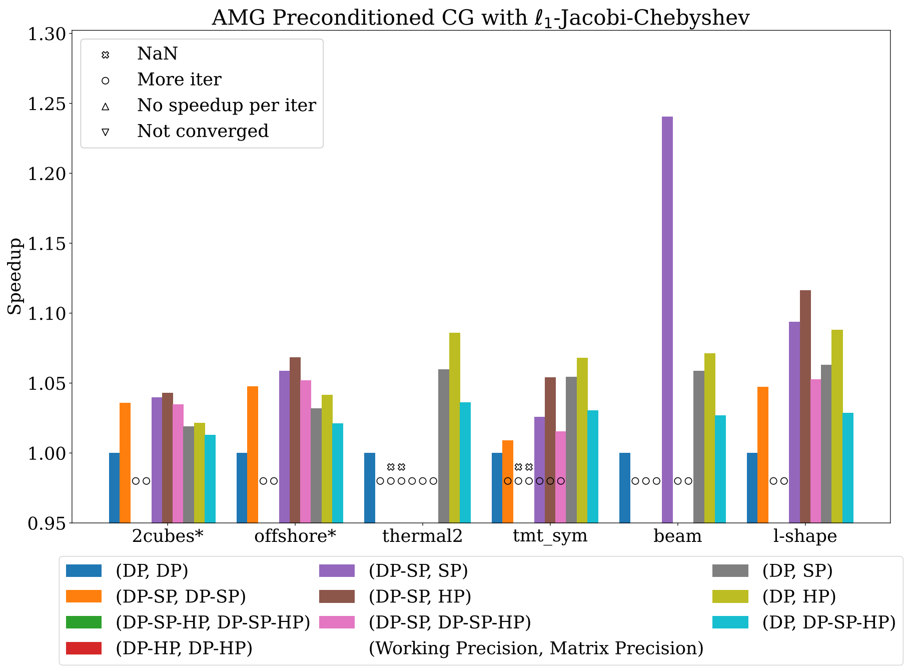


Figure 6.10: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration. Results are for one GCD of MI250X. * denotes the matrix was scaled prior to solving. The detailed meanings of labels are available in Table 6.3.

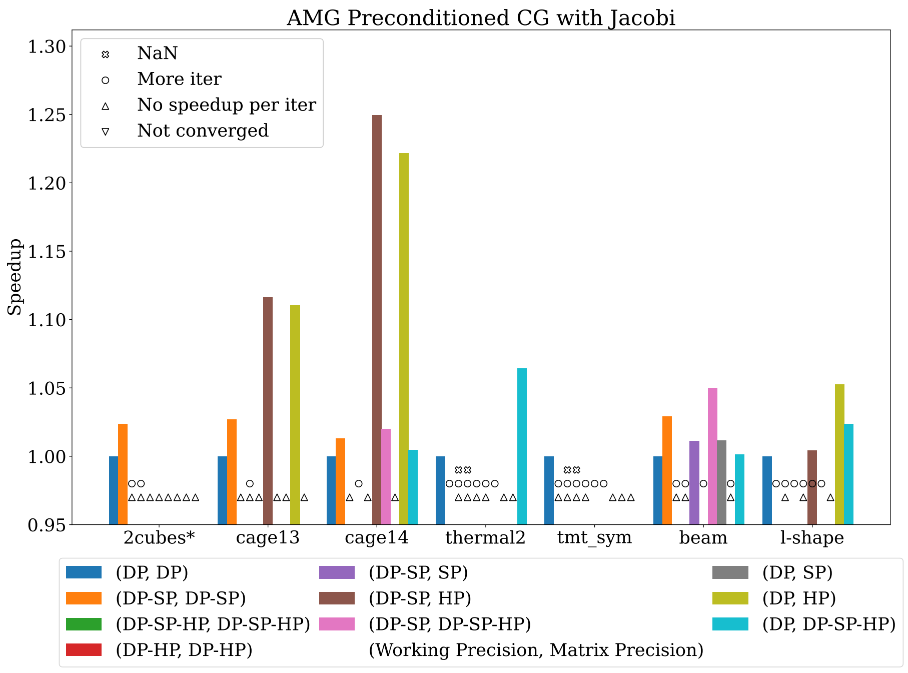


Figure 6.11: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for one tile of Max1550 with the packed half variant SpMV. * denotes the matrix was scaled prior to solving. The detailed meanings of labels are available in Table 6.3.

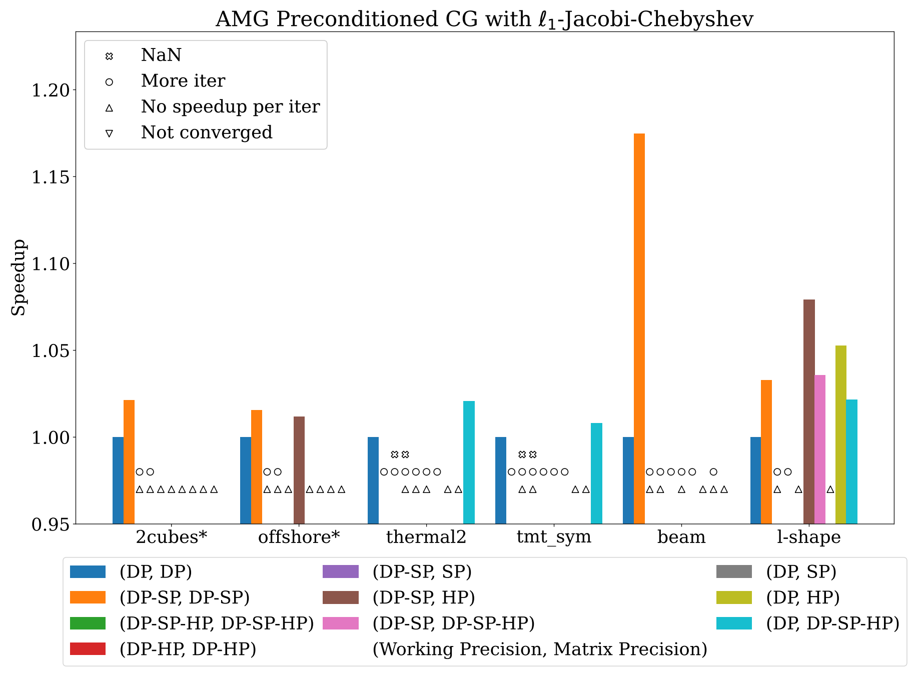


Figure 6.12: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration. Results are for one tile of Max1550 with the packed half variant SpMV. * denotes the matrix was scaled prior to solving. The detailed meanings of labels are available in Table 6.3.

6.5 Adding BFLOAT16 Precision Into the Multigrid Hierarchy

We also add the bfloat16 precision to Ginkgo because the issues discussed in Section 6.3 are mainly coming from the range issue of half precision. The bfloat16 precision gives the same range as the single precision, which avoids underflow/overflow issues encountered when using the half precision. Because bfloat16 has the same memory footprint as half precision, bfloat16 performs similarly to half precision. Some cases need more iterations to converge than complete double precision due to fewer fraction bits with bfloat16. We evaluate the performance of uniform and non-uniform precision settings by replacing half precision with bfloat16 precision. We use BF as the short notation for bfloat16. The experiments test the following precision configurations additionally:

- Uniform Precision: (DP-BF, DP-BF) and (DP-SP-BF, DP-SP-BF)
- Non-Uniform Precision: (DP-SP, BF), (DP-SP, DP-SP-BF), (DP, BF), and (DP, DP-SP-BF)

We also collect the data with uniform level configurations like Table 6.2 in Table 6.4 on H100. The main difference between half precision and bfloat16 precision is that bfloat16 no longer gives NaN for the 2cubes_sphere, offshore, thermal2, and tmt_sym problems. For the 2cubes_sphere, cage13, cage14, and beam problems, the mixed precision configurations with bfloat16 precision use the same number of iterations as the double precision. However, thermal2, tmt_sym, and l-shape need

	Ginkgo's AMG (DP-SP-HP)			Ginkgo's AMG (DP-HP)			Ginkgo's AMG (DP-SP-BF)			Ginkgo's AMG (DP-BF)		
problem	res. norm	#iter	time[ms]	res. norm	#iter	time[ms]	res. norm	#iter	time[ms]	res. norm	#iter	time[ms]
2cubes	NaN	800	493.44	NaN	800	485.06	6.70e-09	20	13.53	1.15e-08	20	13.31
cage13	4.24e-10	11	14.18	6.15e-10	15	17.79	3.67e-10	11	13.88	3.71e-10	11	13.76
cage14	3.73e-10	10	29.04	7.02e-10	13	35.15	3.40e-10	10	29.46	3.42e-10	10	27.60
offshore	NaN	800	706.05	NaN	800	671.65	1416.76	800	713.00	2045.25	800	692.75
thermal2	NaN	800	952.22	NaN	800	919.08	2.80e-06	597	730.73	2.92e-06	635	723.24
tmt_sym	NaN	800	703.4	NaN	800	684.14	9.59e-05	649	593.51	9.87e-05	754	665.21
beam(o3l3)	5.53e-15	86	80.72	7.27e-15	127	117.62	3.15e-15	44	42.90	3.29e-15	44	42.99
l-shape(o3l7)	3.74e-11	800	869.77	2.19e-07	800	868.83	5.01e-14	186	203.06	5.10e-14	192	208.69

Table 6.4: Performance of CG preconditioned with an AMG V-cycle, scalar Jacobi with uniform level precision configurations containing the half precision or bfloat16 precision, on H100. The "packed half" SpMV implementation was used.

more iterations than the double precision due to the lower accuracy of bfloat16, which already showed with (DP-SP) uniform configuration in Table 6.2. Bfloat16 precision gives the same representation range of floating point as single precision, so we do not face the numerical unrecoverable issue with half precision introduced in Section 6.3.

Besides Table 6.4, we also collect the performance overview with all uniform and non-uniform level configurations from the two kinds of smoothers setups in Figures 6.13 and 6.14 on H100, Figures 6.15 and 6.16 on MI250X, and Figures 6.17 and 6.18 on Max1550. We do not pre-scale the 2cube_sphere and offshore problem in these figures, so the uniform levels with half precision do not converge for these problems.

Figures 6.13 and 6.14 show the performance overview without scaling on H100. Bfloat16 and half precision perform similarly if they use the same number of iterations to solve the problem. The similar performance is expected as both precision formats use the same number of bits to represent floating point. Like the uniform level observation, the mixed precision configurations with bloat16 all converge without NaN issue. For beam and l-shape, (DP-SP, DP-SP-BF) needs more iterations, but (DP-SP, DP-SP-HP) does not, which may be related to the fraction bits length between half precision and bfloat16 precision. In Figure 6.14, (DP-SP, DP-SP-HP) gives better performance than (DP-SP, DP-SP-BF). Although bfloat16 precision reduces the concern about underflow and overflow, half precision can still give performance and accuracy benefits when the problem is with sensible accuracy.

We show the performance overview of all configurations, including bfloat16 on MI250X in Figures 6.15 and 6.16. Bfloat16 and half precision still perform similarly when using the same iterations. For the beam problem in Figure 6.16, the cases with more iterations on MI250X are different from the cases on H100 because the AMD GPU supports up to 64 sub-wavefront size and the different 16-bits SpMV implementation that lead to different accumulation results. The thermal2 problem can be solved with bfloat16 uniform level configurations. However, it requires too many additional iterations to beat the performance of the double-precision configuration.

We collect the performance data in the overview plot Figure 6.17 for scalar Jacobi smoothers and Figure 6.18 for ℓ_1-Jacobi-Chebyshev smoother on Max1550. We mainly have the speedup from mixed precision with the cage14 problem. It might also be related to the larger kernel execution overhead than the other hardware such that we need larger speedup from mixed precision SpMV to show a noticeable speedup in total runtime.

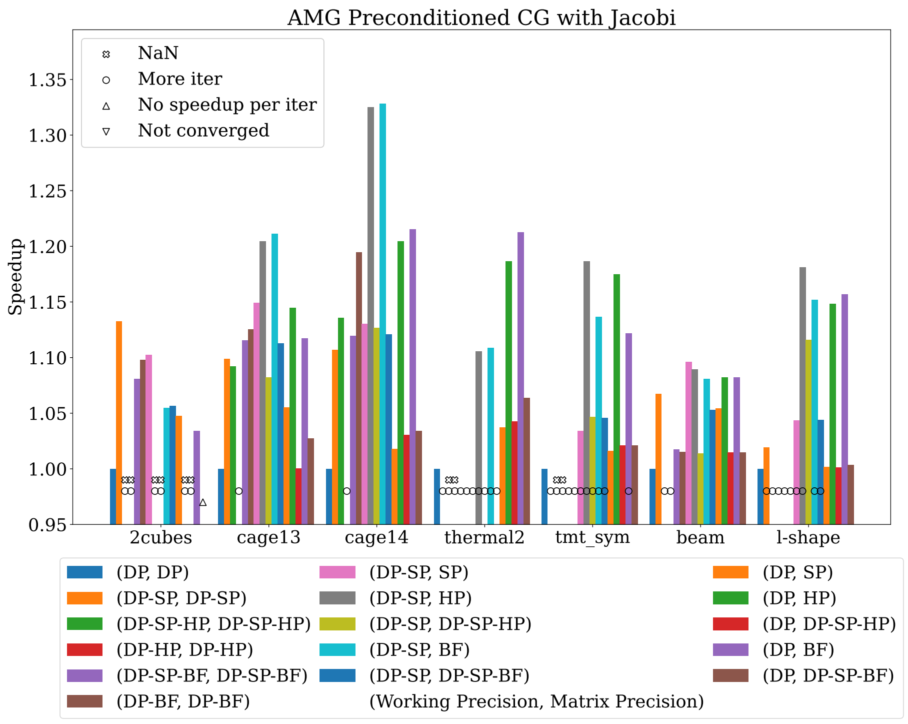


Figure 6.13: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for H100 with the packed half SpMV variant. The detailed meanings of labels are available in Table 6.3.

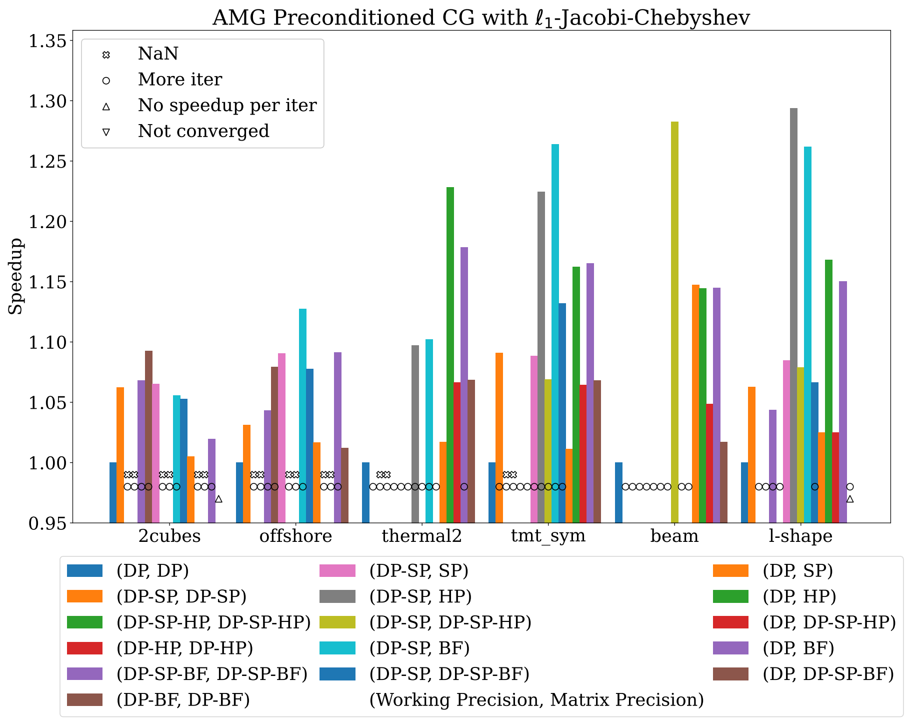


Figure 6.14: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration. Results are for H100 with the packed half SpMV variant. The detailed meanings of labels are available in Table 6.3.

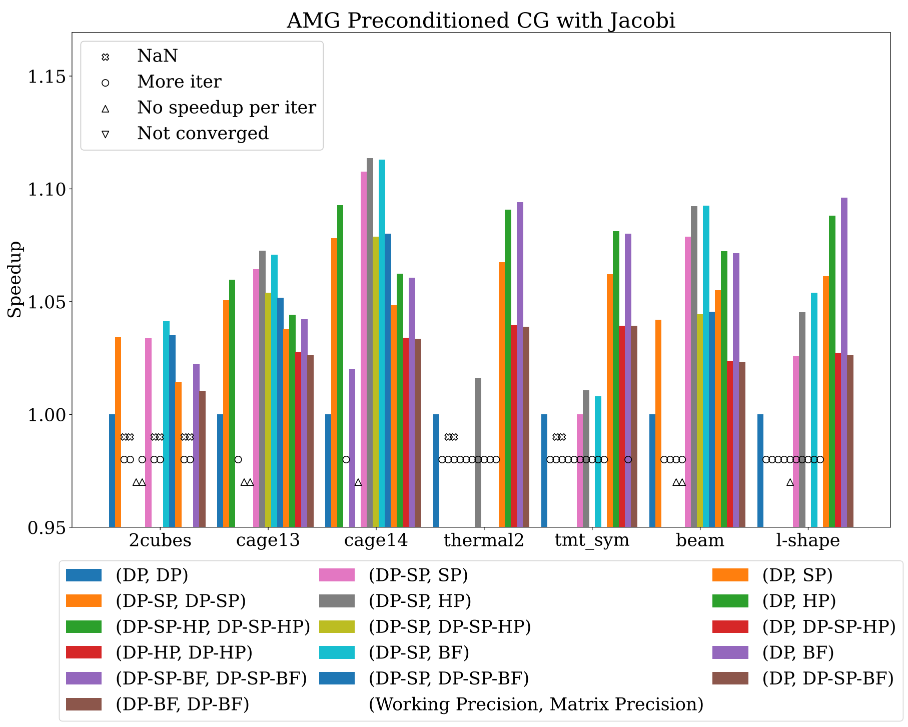


Figure 6.15: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for one GCD of MI250X. The detailed meanings of labels are available in Table 6.3.

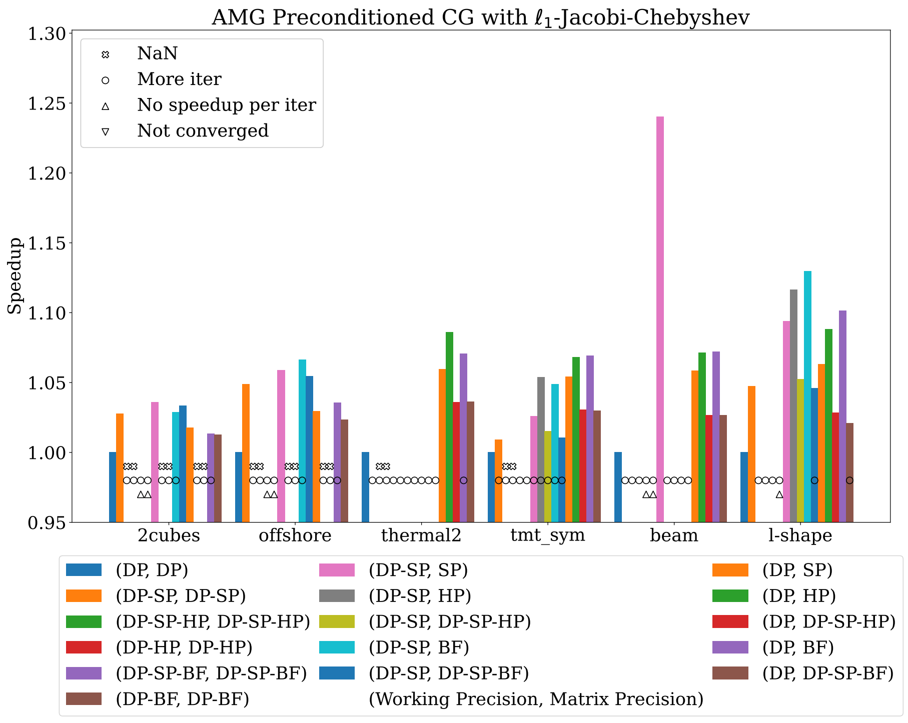


Figure 6.16: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration. Results are for one GCD of MI250X. The detailed meanings of labels are available in Table 6.3.

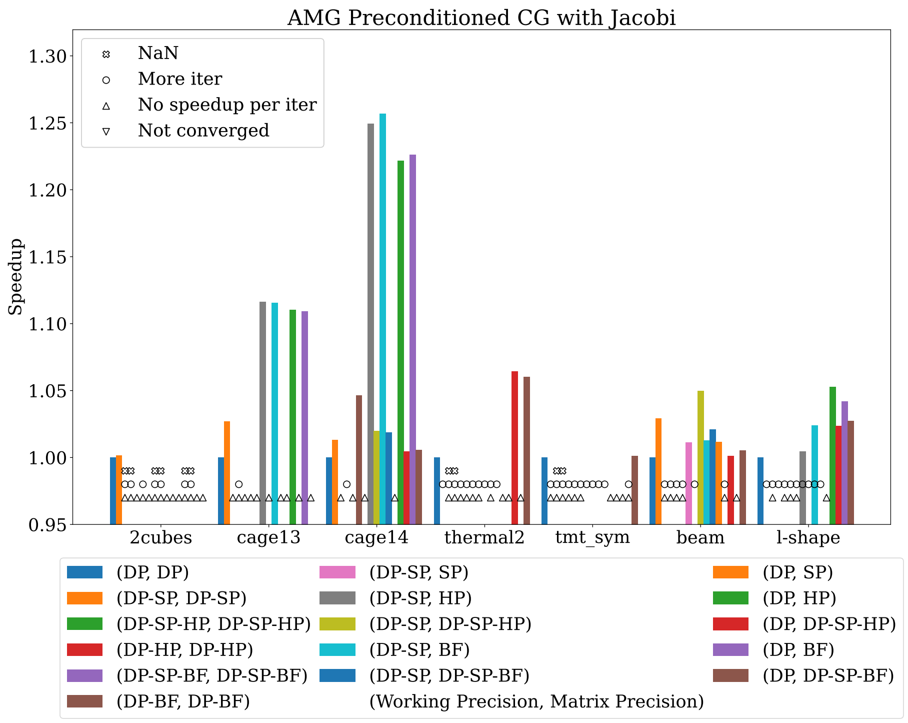


Figure 6.17: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and scalar Jacobi smoother configuration. Results are for one tile of Max1550 with the packed half variant SpMV. The detailed meanings of labels are available in Table 6.3.

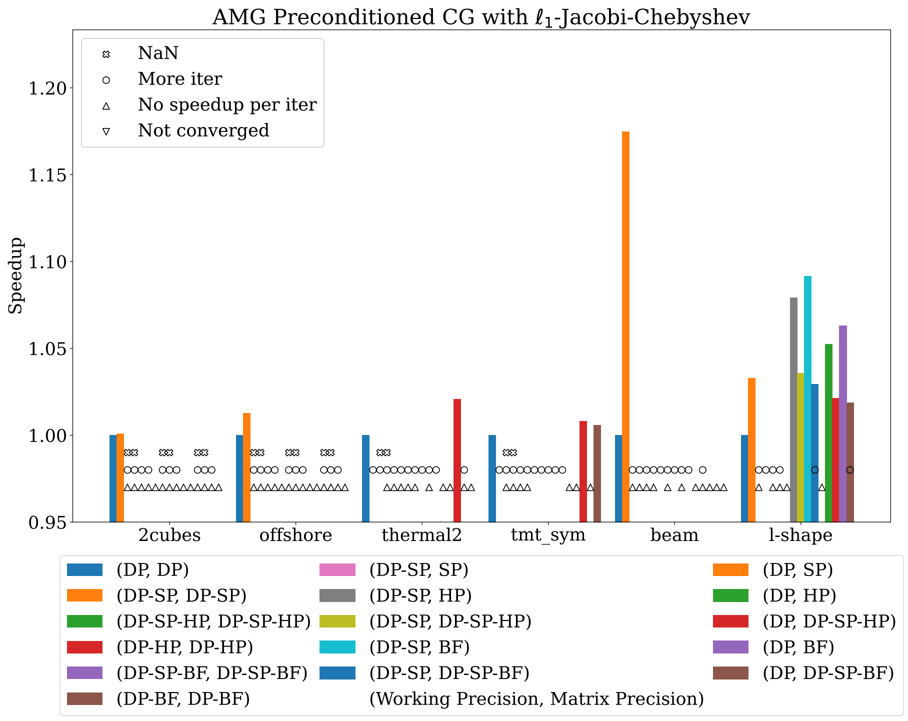


Figure 6.18: Speedup in terms of total solve time for CG with AMG V-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration. Results are for one tile of Max1550 with the packed half variant SpMV. The detailed meanings of labels are available in Table 6.3.

6.6 Standalone Multigrid Solver

This section uses Ginkgo's AMG as a standalone solver on H100. We collect the performance and convergence data in Figure 6.19 and Figure 6.20. We use 1e-9 for the absolute residual norm stopping criterion, and the other parameters are the same as the preconditioner settings in Section 6.1. Four cases converge in complete double precision configuration (i.e. (DP, DP)): cubes_sphere, cage13, cage14, and beam. Thus, we show these four cases in the figures of this section with different precision configurations. Because we already consider bfloat16 as an option for precision, we do not apply the scaling method such that the configuration with half precision results in NaN issue with the 2cubes_sphere problem. The uniform level configuration (DP-HP, DP-HP) is generally poor for convergence, but it is expected, as we discussed the challenges of using half precision previously: the cage13 and cage14 problems need more iterations to recover the rounding effects, and the 2cubes_sphere and beam problems can not converge in the (DP-HP, DP-HP) configuration. As we observe in the experiments using Ginkgo's AMG as a preconditioner, the non-uniform configuration with bfloat16/half precision matrices usually performs the best except for the beam problem. Because there is no additional workload outside of multigrid, the standalone AMG shows higher speedup than the preconditioned CG. The cage14 problem achieves up to 1.45x speedup with (DP-SP, HP) and (DP-SP, BF) configurations. (DP-HP, DP-HP) requires more iterations than (DP-BF, DP-BF) with the cage13 and cage14 problems also due to the underflow issue. However, for the beam problem, (DP, BF) and (DP-SP, BF) show more iterations than (DP, HP) and (DP-SP, HP) in Figure 6.20. From the experiments, using a double precision smoother only at the finest level but the other component in lower precision is sufficient to reach the desired stopping criterion without any outer CG solver or iterative refinement.

6.7 Other Multigrid Cycles

We observe higher speedups when using Ginkgo's mixed precision AMG as a standalone solver instead of embedding it into a CG iterative solver. This is expected as embedding AMG into a CG as a preconditioner reduces the effect of using low precision inside the lower multigrid levels compared to the more significant portion of double precision computations. Another speedup limitation in the V-cycle is the reduced problem size in coarse levels. Even if we reach a good speedup in the coarse levels, the speedup contribution may not be noticeable because the coarse level takes little time overall. Some applications use the W-cycle or F-cycle rather than V-cycle for better convergence. W-cycle(Figure 2.15b) or F-cycle use the coarse level more often than V-cycle(Figure 2.15a) such that the coarse levels play a more important role than the V-cycle in the overall performance. To assess the mixed precision benefit in the cycles other than the V-cycle, we try the same preconditioned CG configuration of Section 6.1 on a small W-cycle with four levels. In Figures 6.21 and 6.22, we can get up to 1.55x speedup with Jacobi smoothers and 1.35x speedup with ℓ_1-Jacobi Chebyshev smoothers on H100. As with the V-cycle, when separating the precision of working vectors and matrices at a level, we can use lower precision for the matrix at the finest level to enable even higher speedups.

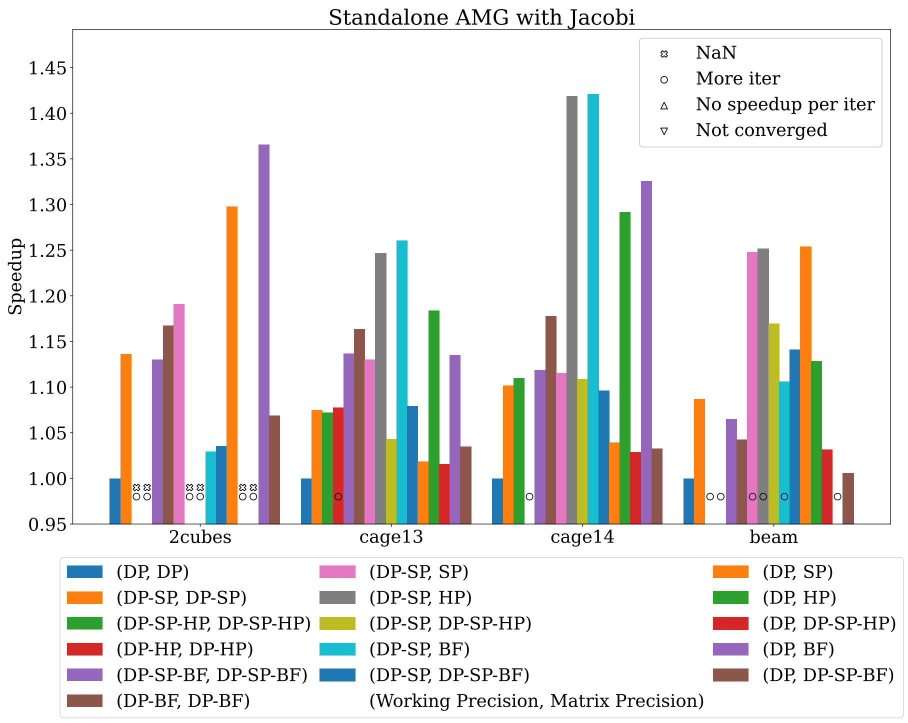


Figure 6.19: Speedup in terms of total solve time for a standalone V-cycle AMG solver (scalar Jacobi smoother configuration) on H100, using packed half SpMV. The detailed meanings of labels are available in Table 6.3.

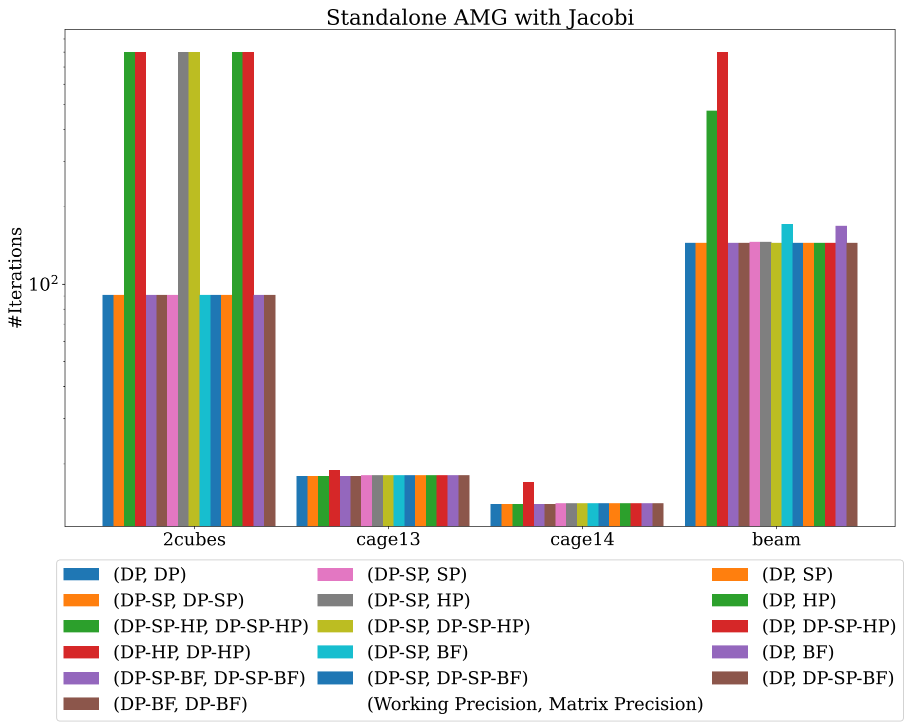


Figure 6.20: Total iterations for a standalone V-cycle AMG solve (scalar Jacobi smoother configuration) on H100, using packed half SpMV.

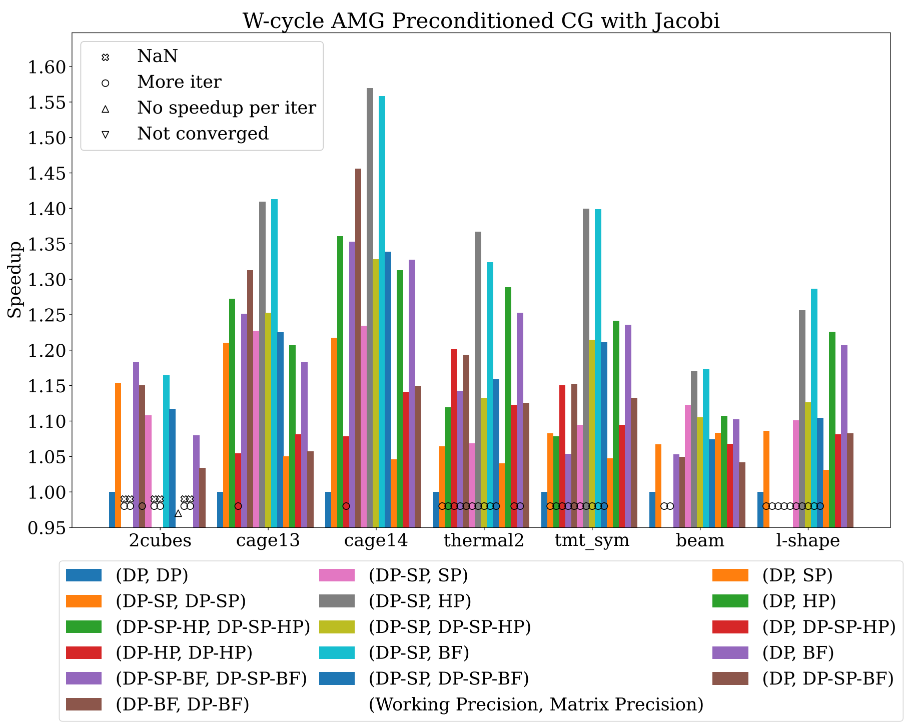


Figure 6.21: Speedup in terms of total solve time for CG with AMG W-cycle preconditioning and scalar Jacobi smoother configuration with 4 levels. Results are for H100 with the packed half SpMV variant. The detailed meanings of labels are available in Table 6.3.

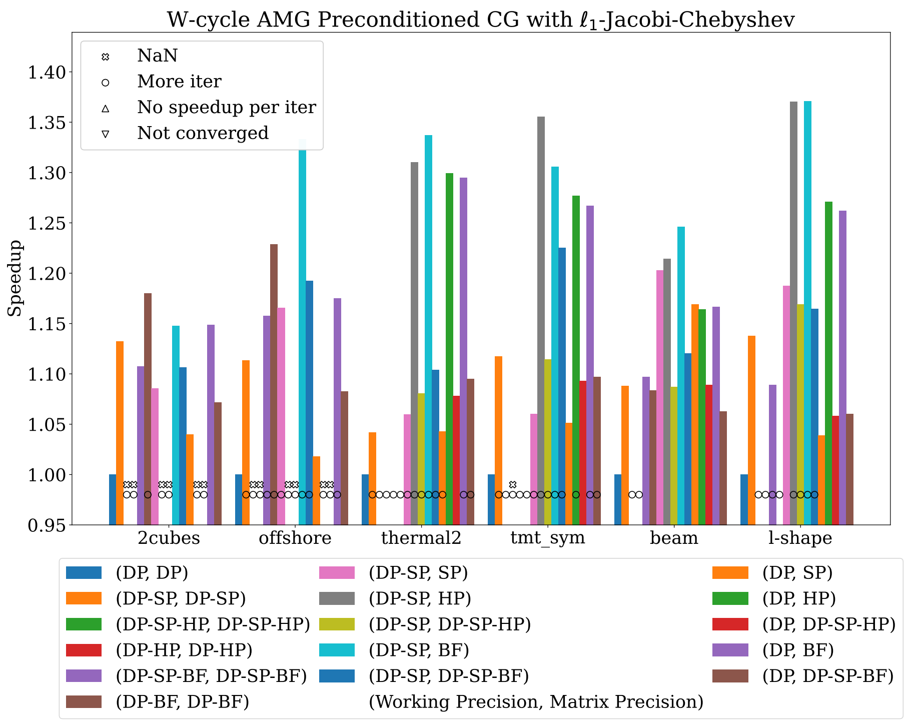


Figure 6.22: Speedup in terms of total solve time for CG with AMG W-cycle preconditioning and ℓ_1-Jacobi-Chebyshev smoother configuration with 4 levels. Results are for H100 with the packed half SpMV variant. The detailed meanings of labels are available in Table 6.3.

6.8 Summary

In this chapter, we used the mixed precision SpMV in Chapter 4 and the flexible design in Chapter 5 to implement and evaluate a mixed precision algebraic multigrid method which allows mixing usage of double precision, single precision, half precision, and bfloat16 precision. When using AMG as a preconditioner, we can use single or even half/bfloat16 precision to reach higher performance without losing convergence in some cases. In uniform configurations handling all components of a multigrid level in the same format, mixed precision AMG using single precision for subsequent levels is a good option. Using half precision results in a significant amount of underflow/overflow problems. We proposed the non-uniform level configurations, which can set the precision for the matrices and working vectors separately. The non-uniform level configurations avoid the underflow issue from half precision because the working vectors with higher precision can properly handle the small values.

Thanks to the bfloat16 precision, we can reduce the underflow/overflow issues. Scaling or decoupling the precision at the same level can be used to overcome the convergence issues introduced by the half precision. By using higher precision for working vectors in non-uniform configurations, we observe that mixed precision with single/half/bfloat16 precision is a viable option. Using higher precision in the working vector maintains the convergence without losing the benefit from mixed precision much. Some cases that are sensitive to rounding errors in the lower levels may still prefer the half precision over the bfloat16 precision because half precision can be more accurate than the bfloat16 precision. Accelerators and ecosystems may only have native support for half precision because it is from the IEEE standard. Without native support from hardware and ecosystems, the lack of optimization with the bfloat16 precision may lead to low performance. We also demonstrate the portability of the developed mixed precision AMG by running performance and convergence experiments on GPUs from different vendors: AMD MI250X(1 GCD), Intel Max1550 (1 tile), and NVIDIA H100(PCIE).

7. Conclusion

We designed and developed the first open-source high-performance mixed-precision AMG, which is portable across GPUs from AMD, Intel, and NVIDIA. We reach this target from the aspects of portability, high performance, and flexibility. We developed mixed precision AMG on GINKGO because GINKGO enforces several principles to ensure software sustainability. To achieve portability, we extend GINKGO'S support from NVIDIA GPUs to AMD, Intel, and NVIDIA GPUs in Chapter 3, which allows developers to use the native vendors' language to optimize the GPU kernels. As SPMV is the backbone of AMG, we design and implement high-performance SPMV routines that are competitive with vendors' libraries. Furthermore, we provide a comprehensive set of mixed-precision SpMV allowing any precision formats input in Chapter 4. We design platform-portable and high-performance AMG in Chapter 5, which shows competitive performance against the state-of-the-art library NVIDIA's AmgX on NVIDIA GPUs. Combining these building blocks, we develop the high-performance mixed-precision AMG for scientific applications, which is available on AMD, Intel, and NVIDIA GPUs in Chapter 6. Although mixed-precision AMG introduces numerical challenges for some problems, we discuss several strategies to overcome these. We demonstrate that using low precision in the AMG hierarchy can improve the performance without hurting the accuracy as a preconditioner or solver.

In the future, we can push these works further in several directions. As the RISC-V processor receives increasing interest, GINKGO can use the extensible design to add RISC-V support or use the existing SYCL backend but optimize the kernels for the RISC-V processor. We can implement a new matrix format, which is suitable for application usage, or try to use tensor cores for sparse matrix formats. Continuing to improve the SPMV performance and adapt to the new devices is always in GINKGO'S mind. The mixed precision AMG design does not restrict the coarsening method, so we can also add more algebraic methods or even geometric multigrid if necessary. With a numerical analysis, we may find some low-overhead algorithms to decide the precision for different levels on a given matrix and algorithms. Because the coarse matrix might not utilize the accelerators fully, we can consider additive multigrid like BFX [BPX90] to apply the smoother concurrently for different levels. Another interesting direction is to use asynchronous concepts in the multigrid design [WC19].

Relevant publications of the author

[Ali+21] José Ignacio Aliaga, Hartwig Anzt, Enrique S. Quintana-Ortí, Andrés E. Tomás, and Yuhsiang M. Tsai. "Balanced and Compressed Coordinate Layout for the Sparse Matrix-Vector Product on GPUs". In: *Euro-Par 2020: Parallel Processing Workshops*. Ed. by Bartosz Balis, Dora B. Heras, Laura Antonelli, Andrea Bracciali, Thomas Gruber, Jin Hyun-Wook, Michael Kuhn, Stephen L. Scott, Didem Unat, and Roman Wyrzykowski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 83–95. ISBN: 978-3-030-71593-9. DOI: 10.1007/978-3-030-71593-9_7.

[Anz+19a] Hartwig Anzt, Yen-Chen Chen, Terry Cojean, Jack Dongarra, Goran Flegar, Pratik Nayak, Enrique S. Quintana-Ortí, Yuhsiang M. Tsai, and Weichung Wang. "Towards Continuous Benchmarking: An Automated Performance Evaluation Framework for High Performance Software". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 1–11. ISBN: 978-1-4503-6770-7. DOI: 10.1145/3324989.3325719.

[Anz+20a] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. "Ginkgo: A High Performance Numerical Linear Algebra Library". In: *Journal of Open Source Software* 5.52 (Aug. 2020), p. 2260. ISSN: 2475-9066. DOI: 10.21105/joss.02260.

[Anz+20b] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. "Load-Balancing Sparse Matrix Vector Product Kernels on GPUs". In: *ACM Transactions on Parallel Computing* 7.1 (Mar. 2020), 2:1–2:26. ISSN: 2329-4949. DOI: 10.1145/3380930.

[Anz+20c] Hartwig Anzt, Yuhsiang M. Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. "Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations". In: *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Nov. 2020, pp. 26–38. DOI: 10.1109/PMBS51919.2020.00009.

[Anz+22] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. "GINKGO: A Modern Linear Operator Algebra Framework for High Performance Computing". In: *ACM*

Transactions on Mathematical Software 48.1 (Feb. 2022), 2:1–2:33. ISSN: 0098-3500. DOI: 10.1145/3480935.

[CTA22] Terry Cojean, Yu-Hsiang Mike Tsai, and Hartwig Anzt. “Ginkgo—A Math Library Designed for Platform Portability”. In: *Parallel Computing* 111 (July 2022), p. 102902. ISSN: 0167-8191. DOI: 10.1016/j.parco.2022.102902.

[TBA23a] Yu-Hsiang Mike Tsai, Natalie Beams, and Hartwig Anzt. “Mixed Precision Algebraic Multigrid on GPUs”. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2023, pp. 113–125. ISBN: 978-3-031-30442-2. DOI: 10.1007/978-3-031-30442-2_9.

[TBA23b] Yu-Hsiang Mike Tsai, Natalie Beams, and Hartwig Anzt. “Three-Precision Algebraic Multigrid on GPUs”. In: *Future Generation Computer Systems* 149 (Dec. 2023), pp. 280–293. ISSN: 0167-739X. DOI: 10.1016/j.future.2023.07.024.

[TCA20] Yuhsiang M. Tsai, Terry Cojean, and Hartwig Anzt. “Sparse Linear Algebra on AMD and NVIDIA GPUs – The Race Is On”. In: *High Performance Computing*. Ed. by Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 309–327. ISBN: 978-3-030-50743-5. DOI: 10.1007/978-3-030-50743-5_16.

[TCA22] Yuhsiang M. Tsai, Terry Cojean, and Hartwig Anzt. “Porting Sparse Linear Algebra to Intel GPUs”. In: *Euro-Par 2021: Parallel Processing Workshops*. Ed. by Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 57–68. ISBN: 978-3-031-06156-1. DOI: 10.1007/978-3-031-06156-1_5.

[TCA23] Yu-Hsiang M. Tsai, Terry Cojean, and Hartwig Anzt. “Providing Performance Portable Numerics for Intel GPUs”. In: *Concurrency and Computation: Practice and Experience* 35.20 (2023), e7400. ISSN: 1532-0634. DOI: 10.1002/cpe.7400.

[Tsa+21] Yuhsiang M. Tsai, Terry Cojean, Tobias Ribizel, and Hartwig Anzt. “Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP”. In: *Euro-Par 2020: Parallel Processing Workshops*. Ed. by Bartosz Balis, Dora B. Heras, Laura Antonelli, Andrea Bracciali, Thomas Gruber, Jin Hyun-Wook, Michael Kuhn, Stephen L. Scott, Didem Unat, and Roman Wyrzykowski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 109–121. ISBN: 978-3-030-71593-9. DOI: 10.1007/978-3-030-71593-9_9.

[Tsa+22] Yu-Hsiang Mike Tsai, Pratik Nayak, Edmond Chow, and Hartwig Anzt. “Implementing Asynchronous Jacobi Iteration on GPUs”. In: *2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH)*. Nov. 2022, pp. 1–9. DOI: 10.1109/ScalAH56622.2022.00006.

Bibliography

[Abd+21] Ahmad Abdelfattah et al. "A Survey of Numerical Linear Algebra Methods Utilizing Mixed-Precision Arithmetic". In: *The International Journal of High Performance Computing Applications* 35.4 (July 2021), pp. 344–369. ISSN: 1094-3420. DOI: 10.1177/10943420211003313.

[ACK19] Hartwig Anzt, Terry Cojean, and Eileen Kühn. "Towards a New Peer Review Concept for Scientific Computing Ensuring Technical Quality, Software Sustainability, and Result Reproducibility". In: *PAMM* 19.1 (2019), e201900490. ISSN: 1617-7061. DOI: 10.1002/pamm.201900490.

[Agg+21] Isha Aggarwal, Aditya Kashi, Pratik Nayak, Cody J. Balos, Carol S. Woodward, and Hartwig Anzt. "Batched Sparse Iterative Solvers for Computational Chemistry Simulations on GPUs". In: *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. St. Louis, MN, USA: IEEE, Nov. 2021, pp. 35–43. ISBN: 978-1-66541-128-8. DOI: 10.1109/ScalA54577.2021.00010.

[Agg+22] Isha Aggarwal, Pratik Nayak, Aditya Kashi, and Hartwig Anzt. "Preconditioners for Batched Iterative Linear Solvers on GPUs". In: *Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation*. Ed. by Kothe Doug, Geist Al, Swaroop Pophale, Hong Liu, and Suzanne Parete-Koon. Communications in Computer and Information Science. Cham: Springer Nature Switzerland, 2022, pp. 38–53. ISBN: 978-3-031-23606-8. DOI: 10.1007/978-3-031-23606-8_3.

[AMD] AMD. *rocBLAS Contributor's Guide*.

[And+21] R. Anderson et al. "MFEM: A Modular Finite Element Methods Library". In: *Computers & Mathematics with Applications* 81 (2021), pp. 42–74. DOI: 10.1016/j.camwa.2020.06.009.

[And+99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, Jan. 1999. ISBN: 978-0-89871-447-0. DOI: 10.1137/1.9780898719604.

[Anz+19b] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. "ParILUT - A Parallel Threshold ILU for GPUs". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2019, pp. 231–241. DOI: 10.1109/IPDPS.2019.00033.

[Arn+21] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. “The Deal.II Finite Element Library: Design, Features, and Insights”. In: *Computers & Mathematics with Applications*. Development and Application of Open-source Software for Problems with Numerical PDEs 81 (Jan. 2021), pp. 407–422. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2020.02.022.

[ATD14] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. “Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ Formats on NVIDIA GPUs”. In: *University of Tennessee, Tech. Rep. ut-eecs-14-727* (2014).

[Bak+11] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang. “Multigrid Smoothers for Ultraparallel Computing”. In: *SIAM Journal on Scientific Computing* 33.5 (Jan. 2011), pp. 2864–2887. ISSN: 1064-8275. DOI: 10.1137/100798806.

[Bar+17] Roscoe Bartlett et al. “xSDK Foundations: Toward an Extreme-scale Scientific Software Development Kit”. In: *Supercomputing Frontiers and Innovations: an International Journal* 4.1 (Mar. 2017), pp. 69–82. ISSN: 2409-6008. DOI: 10.14529/jsfi170104.

[Bar+94] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 1994. ISBN: 978-0-89871-328-2. DOI: 10.1137/1.9781611971538.

[Bet] *Better Scientific Software (BSSw)*. https://bssw.io/.

[BG09] Nathan Bell and Michael Garland. “Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC ’09*. Portland, Oregon: ACM Press, 2009, p. 1. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654078.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial, Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2000. ISBN: 978-0-89871-462-3. DOI: 10.1137/1.9780898719505.

[Bly20] David Blythe. “The Xe GPU Architecture”. In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. Aug. 2020, pp. 1–27. DOI: 10.1109/HCS49909.2020.9220591.

[BPR96] Ronald F Boisvert, Roldan Pozo, and Karin A Remington. *The Matrix Market Exchange Formats: Initial Design*. Vol. 5935. Citeseer, 1996.

[BPX90] James H. Bramble, Joseph E. Pasciak, and Jinchao Xu. “Parallel Multilevel Preconditioners”. In: *Mathematics of Computation* 55.191 (1990), pp. 1–22. ISSN: 0025-5718. DOI: 10.2307/2008789. JSTOR: 2008789.

[CETS14] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns”. In: *Journal of Parallel and Distributed Computing*. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing 74.12 (Dec. 2014), pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[Cma] *CMake - Upgrade Your Software Build System*. https://cmake.org/.

[Cod] *Codecov - The Leading Code Coverage Solution*. https://about.codecov.io/.

[Coo] *Cooperative Groups: Flexible CUDA Thread Programming*. https://developer.nvidia.com/blog/cooperative-groups/. Oct. 2017.

[CP14] Victoria Caparrós Cabezas and Markus Püschel. “Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints”. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2014, pp. 222–231. DOI: 10.1109/IISWC.2014.6983061.

[Dal+15] Steven. Dalton, Sean. Baxter, Duane. Merrill, Luke. Olson, and Michael. Garland. “Optimizing Sparse Matrix Operations on GPUs Using Merge Path”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 407–416.

[Dem] James Demmel. *CS267: Supplementary Notes on Floating Point*. https://people.eecs.berkeley.edu/~demmel/cs267/lecture21/lecture21.html.

[DH11] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Transactions on Mathematical Software* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[Dil+17] Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A. Saurous. *TensorFlow Distributions*. Nov. 2017. DOI: 10.48550/arXiv.1711.10604. arXiv: 1711.10604 [cs, stat].

[Dox] *Doxygen: Doxygen*. https://www.doxygen.nl/.

[DSYH06] Hans De Sterck, Ulrike Meier Yang, and Jeffrey J. Heys. “Reducing Complexity in Parallel Algebraic Multigrid Preconditioners”. In: *SIAM Journal on Matrix Analysis and Applications* 27.4 (Jan. 2006), pp. 1019–1039. ISSN: 0895-4798. DOI: 10.1137/040615729.

[E4s] *E4S - Home*. https://e4s-project.github.io/.

[EHANN22] Abdeselam El Haman Abdeselam, Artem Napov, and Yvan Notay. “Porting an Aggregation-Based Algebraic Multigrid Method to GPUs”. In: *ETNA - Electronic Transactions on Numerical Analysis* 55 (2022), pp. 687–705. ISSN: 1068-9613, 1068-9613. DOI: 10.1553/etna_vol55s687.

[Eva] “Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream”. In: *International Journal of Computational Science and Engineering* 17.3 (Jan. 2018), pp. 247–262. ISSN: 1742-7185.

[FA17] Goran Flegar and Hartwig Anzt. “Overcoming Load Imbalance for Irregular Sparse Matrices”. In: *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. IA3’17. New York, NY, USA: ACM, 2017, 2:1–2:8. ISBN: 978-1-4503-5136-2.

[Fis+22] Paul Fischer, Stefan Kerkemeier, Misun Min, Yu-Hsiang Lan, Malachi Phillips, Thilina Rathnayake, Elia Merzari, Ananias Tomboulides, Ali Karakus, Noel Chalmers, and Tim Warburton. “NekRS, a GPU-accelerated Spectral Element Navier–Stokes Solver”. In: *Parallel Computing* 114 (Dec. 2022), p. 102982. ISSN: 0167-8191. DOI: 10.1016/j.parco.2022.102982.

[Fle+21] Goran Flegar, Hartwig Anzt, Terry Cojean, and Enrique S. Quintana-Ortí. “Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software”. In: *ACM Transactions on Mathematical Software* 47.2 (Apr. 2021), 14:1–14:28. ISSN: 0098-3500. DOI: 10.1145/3441850.

[FS14] Robert D. Falgout and Jacob B. Schroder. “Non-Galerkin Coarse Grids for Algebraic Multigrid”. In: *SIAM Journal on Scientific Computing* 36.3 (Jan. 2014), pp. C309–C334. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/130931539.

[FY02] Robert D. Falgout and Ulrike Meier Yang. “Hypre: A Library of High Performance Preconditioners”. In: *Computational Science — ICCS 2002*. Ed. by Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 632–641. ISBN: 978-3-540-47789-1. DOI: 10.1007/3-540-47789-6_66.

[GAQ23] Thomas Grützmacher, Hartwig Anzt, and Enrique S. Quintana-Ortí. “Using Ginkgo’s Memory Accessor for Improving the Accuracy of Memory-Bound Low Precision BLAS”. In: *Software: Practice and Experience* 53.1 (2023), pp. 81–98. ISSN: 1097-024X. DOI: 10.1002/spe.3041.

[Gar+22] David J Gardner, Daniel R Reynolds, Carol S Woodward, and Cody J Balos. “Enabling New Flexibility in the SUNDIALS Suite of Nonlinear and Differential/Algebraic Equation Solvers”. In: *ACM Transactions on Mathematical Software (TOMS)* (2022). DOI: 10.1145/3539801.

[Goo] *GoogleTest*. Google. Oct. 2023.

[Gro+16] Max Grossman, Christopher Thiele, Mauricio Araya-Polo, Florian Frank, Faruk O. Alpak, and Vivek Sarkar. “A Survey of Sparse Matrix-Vector Multiplication Performance on Large Matrices”. In: *CoRR* (2016). arXiv: 1608.00636.

[Hea18] Michael T. Heath. *Scientific Computing*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, Nov. 2018. ISBN: 978-1-61197-557-4. DOI: 10.1137/1.9781611975581.

[HH05] Desmond J. Higham and Nicholas J. Higham. *Matlab Guide*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2005. ISBN: 978-0-89871-578-1. DOI: 10.1137/1.9780898717891.

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2002. ISBN: 978-0-89871-521-7. DOI: 10.1137/1.9780898718027.

[Hin+05] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. “SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers”. In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 363–396. DOI: 10.1145/1089014.1089020.

[Hon+11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. “Accelerating CUDA Graph Algorithms at Maximum Warp”. In: *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011.* 2011, pp. 267–276.

[Hon+19] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. “Adaptive Sparse Tiling for Sparse Matrix Multiplication”. In: *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019.* 2019, pp. 300–314.

[Hon+22] Neil P. Chue Hong et al. “FAIR Principles for Research Software (FAIR4RS Principles)”. In: (Mar. 2022). DOI: 10.15497/RDA00065.

[HPZ19] Nicholas J. Higham, Srikara Pranesh, and Mawussi Zounon. “Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems”. In: *SIAM Journal on Scientific Computing* 41.4 (Jan. 2019), A2536–A2551. ISSN: 1064-8275. DOI: 10.1137/18M1229511.

[HY02] Van Emden Henson and Ulrike Meier Yang. “BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner”. In: *Applied Numerical Mathematics*. Developments and Trends in Iterative Methods for Large Systems of Equations - in Memorium Rudiger Weiss 41.1 (Apr. 2002), pp. 155–177. ISSN: 0168-9274. DOI: 10.1016/S0168-9274(01)00115-5.

[Iee] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[IPS14] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. “Cache-Aware Roofline Model: Upgrading the Loft”. In: *IEEE Computer Architecture Letters* 13.1 (Jan. 2014), pp. 21–24. ISSN: 1556-6064. DOI: 10.1109/L-CA.2013.6.

[Jso] *JSON.* https://www.json.org/json-en.html.

[Kas+22] Aditya Kashi, Pratik Nayak, Dhruva Kulkarni, Aaron Scheinberg, Paul Lin, and Hartwig Anzt. “Batched Sparse Iterative Solvers on GPU for the Collision Operator for Fusion Plasma Simulations”. In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* May 2022, pp. 157–167. DOI: 10.1109/IPDPS53621.2022.00024.

[Kas+23] Aditya Kashi, Pratik Nayak, Dhruva Kulkarni, Aaron Scheinberg, Paul Lin, and Hartwig Anzt. “Integrating Batched Sparse Iterative Solvers for the Collision Operator in Fusion Plasma Simulations on GPUs”. In: *Journal of Parallel and Distributed Computing* 178 (Aug. 2023), pp. 69–81. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2023.03.012.

[KC17] Elias Konstantinidis and Yiannis Cotronis. “A Quantitative Roofline Model for GPU Kernel Performance Estimation Using Micro-Benchmarks and Hardware Metric Profiling”. In: *Journal of Parallel and Distributed Computing* 107 (Sept. 2017), pp. 37–56. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2017.04.002.

[Kha20] Paresh Kharya. *NVIDIA Blogs: TensorFloat-32 Accelerates AI Training HPC Upto 20x*. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/. May 2020.

[Kre+14] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. “A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units”. In: *SIAM Journal on Scientific Computing* 36.5 (Jan. 2014), pp. C401–C423. ISSN: 1064-8275. DOI: 10.1137/130930352.

[KS01] Arnold Krechel and Klaus Stüben. “Parallel Algebraic Multigrid Based on Subdomain Blocking”. In: *Parallel Computing* 27.8 (July 2001), pp. 1009–1031. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(01)00080-1.

[LM12] Amy N. Langville and Carl D. Meyer. *Google’s PageRank and beyond: The Science of Search Engine Rankings*. Princeton, NJ, USA: Princeton University Press, 2012. ISBN: 0-691-15266-7 978-0-691-15266-0.

[Lub85] M Luby. “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC ’85. New York, NY, USA: Association for Computing Machinery, Dec. 1985, pp. 1–10. ISBN: 978-0-89791-151-1. DOI: 10.1145/22145.22146.

[Mfe] *MFEM: Modular Finite Element Methods [Software]*. https://mfem.org/. DOI: 10.11578/dc.20171025.1248.

[MG16] Duane Merrill and Michael Garland. “Merge-Based Parallel Sparse Matrix-Vector Multiplication”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’16. Piscataway, NJ, USA: IEEE Press, 2016, 58:1–58:12. ISBN: 978-1-4673-8815-3.

[MGG15] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. “High-Performance and Scalable GPU Graph Traversal”. In: *TOPC* 1.2 (2015), 14:1–14:30.

[NA23] Pratik Nayak and Hartwig Anzt. “Utilizing Batched Solver Ideas for Efficient Solution of Non-Batched Linear Systems”. In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2023, pp. 662–665. DOI: 10.1109/IPDPSW59300.2023.00113.

[Nau+15] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka. "AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods". In: *SIAM Journal on Scientific Computing* 37.5 (Jan. 2015), S602–S626. ISSN: 1064-8275. DOI: 10.1137/140980260.

[Not10] Yvan Notay. "An Aggregation-Based Algebraic Multigrid Method". In: *Electronic Transactions on Numerical Analysis* 37 (2010), pp. 123–146.

[NV08] Yvan Notay and Panayot S. Vassilevski. "Recursive Krylov-based Multigrid Cycles". In: *Numerical Linear Algebra with Applications* 15.5 (2008), pp. 473–487. ISSN: 1099-1506. DOI: 10.1002/nla.542.

[NVI] NVIDIA. *CUDA Best Practices.*

[Nvia] *NVIDIA H100 Tensor Core GPU Architecture Overview.* https://resources.nvidia.com/en-us-tensor-core.

[Nvib] *NVIDIA Hopper Architecture In-Depth.* https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/. Mar. 2022.

[Ope] *OpenFOAM | Free CFD Software | The OpenFOAM Foundation.* https://openfoam.org/.

[ope+23] openCARP consortium et al. *openCARP*. 2023. DOI: 10.35097/1027.

[PCW18] Cosmin G. Petra, NaiYuan Chiang, and Jingyi Wang. *HiOp – User Guide*. Tech. rep. LLNL-SM-743591. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2018.

[Pla+21] Gernot Plank*, Axel Loewe*, Aurel Neic*, Christoph Augustin, Yung-Lin (Cary) Huang, Matthias Gsell, Elias Karabelas, Mark Nothstein, Jorge Sánchez, Anton Prassl, Gunnar Seemann*, and Ed Vigmond*. "The openCARP Simulation Environment for Cardiac Electrophysiology". In: *Computer Methods and Programs in Biomedicine* 208 (2021), p. 106223. DOI: 10.1016/j.cmpb.2021.106223.

[Pre] Tom Preston-Werner. *Semantic Versioning 2.0.0.* https://semver.org/.

[QSS07] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical Mathematics*. Vol. 37. Texts in Applied Mathematics. New York, NY: Springer, 2007. ISBN: 978-1-4757-7394-1 978-0-387-22750-4. DOI: 10.1007/b98885.

[Roc] *rocALUTION*. ROCm Software Platform. Nov. 2023.

[RS87] J. W. Ruge and K. Stüben. "4. Algebraic Multigrid". In: *Multigrid Methods*. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 1987, pp. 73–130. ISBN: 978-1-61197-188-0. DOI: 10.1137/1.9781611971057.ch4.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2003. ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.

[Sfi] *SFINAE - Cppreference.Com.* https://en.cppreference.com/w/cpp/language/sfinae.

[SJ22] Alan Smith and Norman James. "AMD Instinct™ MI200 Series Accelerator and Node Architectures". In: *2022 IEEE Hot Chips 34 Symposium (HCS)*. Aug. 2022, pp. 1–23. DOI: 10.1109/HCS55958.2022.9895477.

[Stu] K Stuben. "Algebraic Multigrid (AMG): An Introduction with Applications". In: ().

[Świ+23] Kasia Świrydowicz, Nicholson Koukpaizan, Tobias Ribizel, Fritz Göbel, Shrirang Abhyankar, Hartwig Anzt, and Slaven Peleš. *GPU-Resident Sparse Direct Linear Solvers for Alternating Current Optimal Power Flow Analysis*. Aug. 2023. DOI: 10.48550/arXiv.2306.14337. arXiv: 2306.14337 [cs].

[Syc] *SYCLomatic*. oneAPI-SRC. Oct. 2023.

[Teaa] The Muelu Project Team. *The Muelu Project Website*. https://trilinos.github.io/muelu.html.

[Teab] The Trilinos Project Team. *The Trilinos Project Website*. https://trilinos.github.io.

[Wat04] David S. Watkins. *Fundamentals of Matrix Computations*. John Wiley & Sons, Aug. 2004. ISBN: 978-0-471-46167-8.

[WC19] Jordi Wolfson-Pou and Edmond Chow. "Asynchronous Multigrid Methods". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil: IEEE, May 2019, pp. 101–110. ISBN: 978-1-72811-246-6. DOI: 10.1109/IPDPS.2019.00021.

[Wel+98] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. "A Tensorial Approach to Computational Continuum Mechanics Using Object-Oriented Techniques". In: *Computer in Physics* 12.6 (Nov. 1998), pp. 620–631. ISSN: 0894-1866. DOI: 10.1063/1.168744.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[YB12] Rio Yokota and Lorena A. Barba. "Hierarchical N-body Simulations with Auto-Tuning for Heterogeneous Systems". In: *Computing in Science & Engineering* 14.3 (May 2012), pp. 30–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2012.1. arXiv: 1108.5815 [cs].

Keyword