A Survey of Mixed Precision Multigrid Methods

Jinwen Pan
TUM School of Computation, Information and Technology
Technical University of Munich
Garching, Germany
jinwen.pan@tum.de

***Abstract*—Mixed precision multigrid methods have emerged as a promising technique for solving large-scale linear systems in various scientific and engineering applications by combining low and high precision floating-point formats to expedite calculations while maintaining accuracy. This paper provides a survey of the state-of-the-art in mixed precision multigrid methods, encompassing theoretical and practical dimensions. Recent studies in this field have primarily concerned integrating low precision multigrid techniques into high precision solvers to achieve efficient computations. However, investigations into utilizing different precisions across multiple levels of grid hierarchy to further enhance efficiency have been relatively limited, and thus present a potentially promising direction for future research in this domain.**

***Index Terms*—mixed precision algorithms, multigrid methods, high performance computing**

I. Introduction

With the exponential growth of large-scale linear systems derived from various scientific and engineering domains and the increasing demand for computational speed, traditional high precision algorithms often become computationally prohibitive. In response to these challenges, mixed precision multigrid methods, as an effective approach utilizing mixed precision, have gained significant attention due to their ability to strike a balance between accuracy and computational efficiency. This section primarily draws upon [1] and [2] as key references.

A. Mixed Precision Algorithms

According to N. J. Higham and T. Mary [2], a mixed precision algorithm employs two or more precisions, selected from a limited set of available precisions, for data storage and computation. These typically include half precision, single precision, and double precision, which are supported by hardware, along with quadruple precision, which is supported by software. Similarly, multiprecision or variable precision algorithms refer to the algorithms utilizing one or more arbitrary precisions, which can vary based on the specific problem and are implemented through software. This paper exclusively focuses on mixed precision algorithms.

In simulation programs, the length (in bits) of the utilized floating-point format profoundly influences both the accuracy and execution time of simulations. Higher precision generally leads to higher accuracy. This is primarily because higher precision formats can represent numbers with more significant digits, allowing for a more precise representation of quantities and reducing rounding errors. The adoption of a more finely grained representation enables computations involving extremely large or minuscule values, as well as those with subtle differences, to be more accurately captured.

The major portion of simulation runtime is typically attributed to data communication and computation. Regarding the former, when ignoring memory access latency, there is usually a hardware-independent linear correlation between the communication overhead and the length of the floating-point format. Conversely, the computational overhead of diverse floating-point formats is largely contingent upon the specific characteristics of hardware architectures [1]. Consequently, the strategic integration of lower precision formats such as single precision (32-bit) or half precision (16-bit), where they are deemed sufficient, with conventional higher precision arithmetics, offers a compelling avenue for achieving significant performance enhancements while preserving the overall accuracy of computations.

The applications of mixed precision algorithms span a wide range of scientific and engineering fields. The following examples are presented to illustrate the potential advantages of mixed precision strategies in real-life applications:

- Physical Simulations: Y. Idomura et al. [3] have employed a mixed precision communication-avoiding (CA) Krylov method to accelerate a Gyrokinetic Toroidal 5D Eulerian program which is one of the most computationally intensive programs in fusion science. The utilization of IEEE half precision preconditioner has facilitated 2.8× and 1.9× speedups on Fugaku and Summit respectively compared to the conventional method.
- Climate Modelling and Weather Forecasting: Since 2014 there has been a notable inclination towards the utilization of low precision formats as substitutes for conventional high precision formats within diverse climate models [4]. T. N. Palmer [5] contends that the representation of data in low precision is sufficient to maintain consistency with the underlying observations on which a model is constructed.
- Machine Learning: M. Courbariaux's work [6] can be recognized as one of the pioneering works in deep learning utilizing low precision arithmetic to train and execute neural networks for computational efficiency. More specifically, S. Gupta et al. [7] suggest that networks can be effectively trained using a mere 16-bit fixed-point number representation when employing stochastic rounding. As a reason that the popularity of low preci-

sion is increasing, the notion that precise solutions are not necessarily required in machine learning is further expounded by F. E. Curtis and K. Scheinberg [8] from the perspective of optimization algorithms.

B. Multigrid Methods

Multigrid methods are powerful numerical techniques widely employed in computational mathematics to solve a linear system of equations that arise from partial differential equations (PDEs). As W. Hackbusch [9] elaborates, the fundamental principle underlying multigrid methods involves addressing the problem across multiple grids of varying resolutions. Consequently, the core concept in multigrid methods revolves around grid transfer, which transfers information between grids with different resolutions. Prolongation operators play a crucial role in this process by interpolating the coarse-grid solution to finer grids, thereby enabling a more precise representation of the solution. Similarly, restriction operators reduce the fine-grid solution to coarser grids, preserving essential information while simultaneously reducing computational complexity. These transfer operators enable error propagation and correction from one grid level to another, thus facilitating efficient convergence.

Typically, multigrid methods operate within a cycle, commonly referred to as a V-cycle or a W-cycle, where the solution is progressively refined through iterative steps. The process commences with an initial approximation on the finest grid, and a sequence of smoothing operations, such as Gauss-Seidel or Jacobi iterations, is applied to improve the solution. Upon convergence or the attainment of a desired level of accuracy, the process transitions to the next coarser grid, where the same set of smoothing operations and transfer operations are employed. This fine-to-coarse progression continues until the coarsest grid is reached, following which the solution is interpolated back to the finest grid, further refining the solution. The entire cycle is repeated until the desired level of accuracy is achieved.

Multigrid methods offer several significant advantages in computational simulations. One of their primary strengths lies in their remarkable ability to achieve rapid convergence towards highly accurate solutions because these methods effectively capture both high-frequency and low-frequency components of the solution. Additionally, multigrid methods demonstrate exceptional scalability, making them well-suited for parallel computing architectures and distributed systems. As a result, there is considerable potential in employing low precision multigrid methods as preconditioners for high precision solvers. Moreover, the hierarchical structure of multigrid methods provides the possibility for utilizing different precisions at each level of the multigrid hierarchy, potentially further enhancing their versatility and adaptability [10].

II. IEEE 754 Floating-Point Arithmetic

IEEE 754 Floating-Point Arithmetic is a widely adopted standard for representing and performing arithmetic operations on real numbers in computers. The IEEE 754 standard defines formats for encoding floating-point numbers, specifying their precision, exponent range, and special values such as infinity and NaN (Not-a-Number). It provides a consistent and portable representation across different computer architectures and programming languages. This section is derived from the technical reference manuals of IEEE [11], [12], and N. J. Higham's book [13].

A. IEEE 754 Floating-Point Number Systems

A system of IEEE 754 floating-point numbers can be defined as a finite subset F of $\mathbb{R}$ with four integers: the base $b \geq 2$ (In contemporary computer systems, the base b is typically 2.), the precision p, the minimal exponent e_{min}, and the maximal exponent e_{max}. Specifically, with an exponent $e \in \{e_{min}, ..., e_{max}\}$ and digits $d_0, ..., d_{p-1} \in \{0, ..., b-1\}$, an IEEE 754 floating-point number x can be represented as

$$x = \pm b^e \times \left(d_0 + \frac{d_1}{b} + \frac{d_2}{b^2} + ... + \frac{d_{p-1}}{b^{p-1}} \right). \tag{1}$$

Alternatively, it can also be represented as

$$x = \pm m \times b^{e-p+1}, \tag{2}$$

where the significand $m \in \{0, b, ..., b^p - 1\}$. For normalization, the restrictions $d_0 \in \{1, ..., b-1\}$ and $m \in \left\{b^{p-1}, b^{p-1} + b, ..., b^p - 1\right\}$ should be imposed to the equations above respectively. These normalized numbers are unevenly spaced because of their variable exponents. The largest positive normalized floating-point number is $x_{max} = b^{e_{max}} \times (b - b^{1-p})$ and the smallest positive normalized floating point number is $x_{min} = b^{e_{min}}$. Therefore,

$$b^{e_{min}} \leq |x| \leq b^{e_{max}} \times (b - b^{1-p}). \tag{3}$$

The smallest normalized floating-point number larger than 1 is $x = 1 + b^{1-p}$, and the distance $x - 1 = b^{1-p}$ is defined as the machine epsilon ϵ. Furthermore, $u = \frac{1}{2} b^{1-p}$ is defined as the unit roundoff. The unit roundoff in higher precision formats is smaller, while the unit roundoff in lower precision formats is relatively larger.

IEEE 754 also defines subnormal numbers with $e = e^{e_{min}}$, $d_0 = 0$, and $m \in \left\{0, b, ..., b^{p-1} - 1\right\}$. These numbers are evenly spaced between 0 and the smallest normalized number because of the fixed exponent.

Table I documents most of the currently available IEEE 754 floating-point number systems. The 32-bit IEEE single precision format and the 64-bit IEEE double precision format were introduced in 1985 and gained significant hardware support in the subsequent decade. In 2008, the 16-bit IEEE half precision format was added, exclusively intended for data storage rather than computational operations purposes and primarily supported on GPUs. Additionally, the 128-bit IEEE quadruple precision format was also added in the same year, predominantly supported by software implementations.

In addition to the well-known IEEE arithmetic standards, various other arithmetic formats have emerged to cater to specific applications. For example, the bfloat16 format introduced

TABLE I
IEEE 754 FLOATING-POINT NUMBER SYSTEMS [2]

Format	Sign	Exponent	Significand	e_{min}	e_{max}	Machine Epsilon	x_{min}	x_{max}
FP16	1 bit	5 bits	10 bits	-14	+15	9.76×10^{-4}	6.10×10^{-5}	6.55×10^{4}
FP32	1 bit	8 bits	23 bits	-126	+127	1.19×10^{-7}	1.18×10^{-38}	3.40×10^{38}
FP64	1 bit	11 bits	52 bits	-1022	+1023	2.22×10^{-16}	2.22×10^{-308}	1.80×10^{308}
FP128	1 bit	15 bits	112 bits	-16382	+16383	1.93×10^{-34}	3.36×10^{-4932}	1.19×10^{4932}

by Google has gained widespread recognition. Moreover, a 9-bit floating-point format was implemented by S. O'uchi [14] for the purpose of accelerating deep learning algorithms.

B. Rounding Error Analysis Model

Rounding error analysis is a fundamental aspect of numerical computation that focuses on quantifying the discrepancies between exact mathematical values and their approximate representations due to limited precision in computer arithmetic. A comprehensive examination of existing models pertaining to rounding error is presented in N. J. Higham's book [13]. However, this paper solely introduces and focuses on those models that are directly pertinent to the subject matter at hand.

In this subsection, u always means the unit roundoff of the specific format. For all $x \in \mathbb{R}$, there are a floating-point number $fl(x)$ and a constant $|\delta| \leq u$ such that

$$fl(x) = x(1+\delta). \tag{4}$$

Furthermore, let $\star$ be one of the four fundamental operations addition, subtraction, multiplication, and division. For all x and $y \in \mathbb{R}$, there exist a floating-point number $fl(x \star y)$ and a constant $|\delta| \leq u$ such that

$$fl(x \star y) = (x \star y)(1+\delta). \tag{5}$$

In the context of matrix and vector operations, for all $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{R}^n$, there is a constant vector $\delta \in \mathbb{R}^n$ such that

$$fl(Ax) = Ax + \delta,\ |\delta| \leq \frac{nu}{1-nu}\,|A| \cdot |x|\,. \tag{6}$$

Similarly, for all $A \in \mathbb{R}^{n\times n}$, $x \in \mathbb{R}^n$, and $b \in \mathbb{R}^n$, there is a constant vector $\delta \in \mathbb{R}^n$ such that

$$fl(Ax-b) = Ax - b + \delta,\ |\delta| \leq \frac{(n+1)u}{1-(n+1)u}(|b| + |A| \cdot |x|). \tag{7}$$

The notation $|\cdot|$ represents the transformation of a vector or matrix by replacing its entries with their respective absolute values. Equations (6) and (7) apply to the corresponding entries between the approximate and exact vectors.

C. Matrix-Matrix Multiplication Experiment

Matrix-matrix multiplication is a fundamental operation in numerical algorithms, and its computational efficiency has garnered considerable attention from the academic community. A numerical experiment aimed at optimizing the computational efficiency of matrix-matrix multiplication is conducted and analyzed in this subsection based on three different aspects, with particular emphasis placed on precision considerations to elucidate the potential benefits of employing mixed precision algorithms. The complete project can be accessed in the GitHub repository.

1) Cache Optimization: Cache is a compact and high-speed storage component positioned in closer proximity to the CPU than the main memory, aiming to mitigate the latency involved in memory access. The rationale underlying cache design is grounded in the principle of cache locality, which posits that programs exhibit a propensity to access data and instructions that exhibit spatial or temporal proximity. The optimization of cache locality plays a pivotal role in bolstering program performance, particularly within contemporary computer architectures featuring multi-level cache hierarchies. This optimization entails the strategic structuring of code and data access patterns with the objective of maximizing the utilization of cache memory.

The kernel of a rudimentary implementation of matrix-matrix multiplication is presented in Listing 1, where the three loops are independent and the loop order can be permuted arbitrarily to yield the same computational result. While executing the innermost loop, cache misses are highly probable when retrieving the value of `B[k*n+j]` because the matrix is stored in row-major order, causing each increment of `k` to skip an entire row of the matrix, leading to substantial jumps in memory. However, cache misses are unlikely to

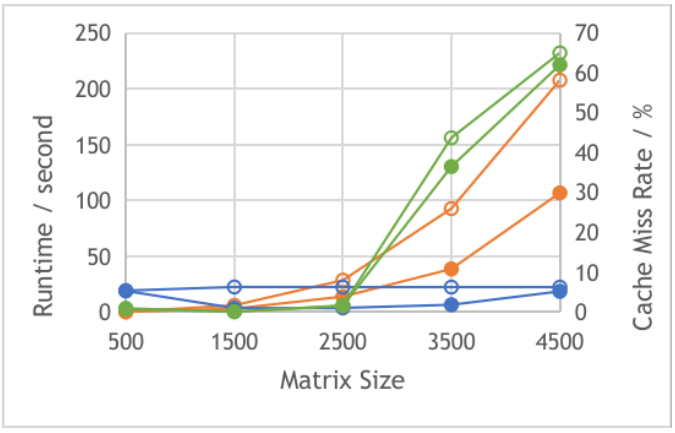

Fig. 1. Cache optimization of matrix multiplication: runtime (orange), L2 cache miss rate (blue), L3 cache miss rate (green); filled symbols: cache-optimized kernel, empty symbols: non-optimized kernel; single precision and `-O1` are utilized.

occur when accessing `C[i*n+j]` and `A[i*n+k]` since these values adhere to row-major order. If the value of `A[i*n+k]` is cached from the previous iteration, the corresponding value read in the present iteration would be derived from an adjacent cache location. Hence, it can be inferred that one cache miss is expected during each iteration of the innermost loop.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i * n + j] += A[i * n + k] * B
                [k * n + j];
        }
    }
}
```

Listing 1. Intuitionistic matrix multiplication kernel

To enhance cache locality, the arrangement of loops is altered, as demonstrated in Listing 2. Employing an analysis similar to the previous one, it is expected that, during each iteration of the innermost loop, cache misses are minimized, resulting in improved performance. As shown in Fig. 1, the runtime has been reduced by approximately half for different problem sizes. This may be attributed to the reduction of L1 cache miss rate (unfortunately, the PAPI hardware counters related to L1 cache are not available on the machine used for the application). On the other hand, the L2 and L3 cache miss rates have not been significantly improved because the size of the matrices is relatively large, and the total number of accesses to the L2 and L3 caches has decreased.

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        for (int j = 0; j < n; j++) {
            C[i * n + j] += A[i * n + k] * B
                [k * n + j];
        }
    }
}
```

Listing 2. Cache-optimized matrix multiplication kernel: compared to the previous kernel, the loop order is modified from $i-j-k$ to $i-k-j$, which takes advantage of cache locality.

To leverage cache locality more effectively, a strategy involves partitioning the original matrices into blocks, performing multiplication on the corresponding blocks, and accumulating the results to yield the final matrix product, which proves particularly advantageous on multicore architectures. Nonetheless, designing such a pattern poses significant challenges and heavily relies on hardware-specific characteristics, such as cache size.

As an alternative to 1D arrays, the utilization of 2D arrays for matrix storage can introduce the issue of memory fragmentation. This arises from the feature of a 2D array, which is structured as an array of arrays, thereby necessitating separate allocations for each row. Consequently, this incurs additional pointer overhead and can potentially increase cache misses, thereby impacting computational performance.

2) Compiler Optimization: Compiler optimization encompasses a repertoire of techniques and transformations implemented by the compiler with the purpose of augmenting execution speed, diminishing code size, and optimizing the utilization of system resources. In this experiment, we investigate a range of compiler flags supported by the Intel C++ compiler, as illustrated in Fig. 2. It is worth noting that comparable outcomes are expected when employing the GNU compiler.

The Intel compiler provides three optimization levels, namely `-O1`, `-O2`, and `-O3`, which offer progressively enhanced performance for compiled code. By default, when a specific level is not specified, `-O2` is employed. The `-O1` optimization level primarily involves the inlining of functions, loop optimization, and basic data flow analysis. Advancing to `-O2` introduces more aggressive but similar optimization techniques. Lastly, `-O3` encompasses a comprehensive range of advanced optimizations, including aggressive loop unrolling, interprocedural optimizations, and extensive inlining. While these optimizations can significantly improve execution speed, they may also result in longer compilation time and larger code size.

In particular, `-O2` and `-O3` optimization levels enable vectorization as a means to enhance program performance by leveraging the inherent parallelism of modern processors. Vectorization entails transforming sequential code into SIMD (Single Instruction, Multiple Data) instructions capable of processing multiple data elements simultaneously. However, not all code can be effectively vectorized. Certain dependencies, control flow patterns, or data access patterns may impose limitations on the efficacy of vectorization. As shown in Fig. 3, `-O3` does not significantly improve performance compared to `-O2`, which may be due to the additional optimization techniques introduced by `-O3` that could potentially lead to more instruction cache misses, register pressure, or other adverse effects for this application.

The utilization of `-ipo` facilitates interprocedural optimization, while the inclusion of `-fno-alias` instructs the compiler to presume the absence of memory locations accessed through multiple pointers, thus enabling more assertive optimization techniques. However, neither of these flags yields a substantial performance improvement when applied to the matrix multiplication application under the default `-O2` level. On the other hand, the adoption of `-xCORE-AVX512` enables the generation of code that capitalizes on broader vector registers and supplementary SIMD operations provided by the AVX-512 instruction set, consequently leading to heightened parallelism and improved performance in vectorized operations.

3) Precision Optimization: The selection of floating-point precision within numerical algorithms exerts a profound influence on both computational efficiency and accuracy. By exploiting the attributes inherent to varying precision levels, substantial improvements in performance can be achieved. Consequently, despite the existence of the IEEE 754 standard, a diverse range of precision formats has been meticulously conceived and implemented to cater to specific applications, even stimulating advancements in hardware development.

As illustrated in Fig. 3, we employ two well-established IEEE standard floating-point formats, namely `float` and

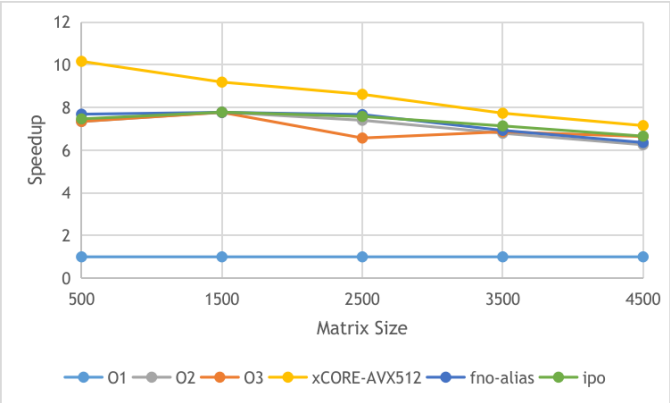

Fig. 2. Compiler optimization of matrix multiplication: `-O1`, `-O2`, and `-O3` are examined individually, while the last three options are examined in combination with default `-O2` respectively; `-O1` is set as the baseline; single precision and cache-optimized kernel are utilized.

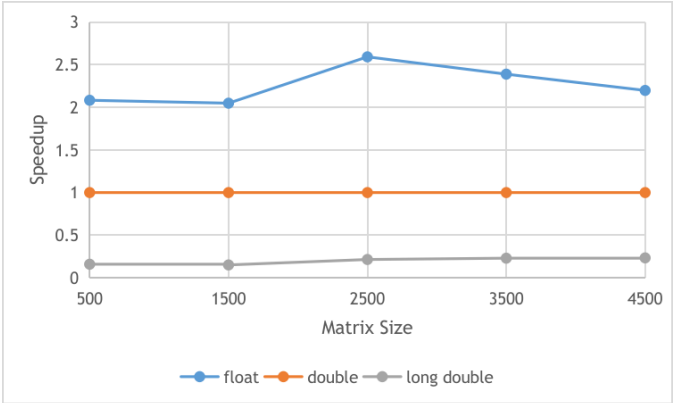

Fig. 3. Precision optimization of matrix multiplication: this experiment is based on the cache-optimized kernel mentioned earlier and the `-O2` compiler optimization level; double precision is set as the baseline.

`double`, alongside the `long double` format to investigate the speed improvements attained in comparison to FP64 arithmetic across various matrix sizes. The performance improvement achieved with the utilization of the `float` precision is striking, exhibiting a consistent and nearly constant speedup factor of 2 across diverse matrix sizes. This outcome can be attributed to the fact that `float` possesses a half bit length compared to `double`. Notably, at a problem size of 2500, the speedup reaches 2.59×, which can be attributed to the interplay between problem size and cache size. In the case of `long double`, the speedup curve exhibits greater stability, hovering around a value of 0.2×. The specific characteristics of long double, such as its size and precision, may vary depending on the programming language, compiler, and hardware platform. In most implementations, long double occupies more memory than double, typically 80 or 120 bits of memory, allowing it to represent a wider range of values with higher precision. This increased precision can be advantageous for applications that require greater numerical accuracy, such as computational fluid dynamics, molecular simulations, and structural analysis. The findings of this numerical experiment offer valuable insights into the prospects of employing mixed precision algorithms in numerical computations in the following manner: employing low precision formats for computational tasks that prioritize efficiency over precision, and conversely employing high precision formats for tasks that demand heightened accuracy. Consequently, this strategy accelerates the overall application without substantially compromising result accuracy.

Notably, the experiment exclusively focuses on sequential optimization techniques for matrix-matrix multiplication. It is essential to acknowledge that many contemporary simulation applications are executed in parallel on multi-core architectures, where the benefits of mixed precision algorithms can be further exploited. However, exploring the implications of parallel execution is beyond the scope of this paper.

III. Performance Evaluation of Mixed Precision Multigrid Methods

Mixed precision multigrid methods have been extensively employed in the field for a considerable period of time. Certain researchers have utilized exclusively low precision multigrid techniques as preconditioners for solving large linear systems of equations. Meanwhile, others have explored the utilization of different precision levels within the multigrid hierarchy to address specific application requirements. This section presents several relevant practical studies, primarily focusing on performance evaluations.

M. Emans and A. van der Meer [15] have implemented four different algebraic multigrid (AMG) algorithms with single precision arithmetic to precondition a standard conjugate gradient algorithm using standard 64-bit double precision format on CPUs. In addition to model linear algebra benchmarks, the authors primarily analyze the performance of the aforementioned techniques in computational fluid dynamics (CFD) simulations. One of the benchmarks simulates a short phase of a complete cycle in a four-cylinder engine, where the cylinder is loaded with cold air. The benchmark results are displayed in Fig. 4. In this simulation, the finest grid consists of 1.4 million unknowns in this linear system of equations, while in a well-functioning multigrid method, the majority of computational work is concentrated on the fine grids. For different solvers, the number of iterations required to terminate the computation is almost the same when using different numbers of processors. Additionally, compared to the double precision solvers, the speedup of the single precision solvers remains between 1.1 and 1.5. Moreover, the parallel efficiency is impacted not only by parallel overhead but also by the slowdown of the fundamental matrix-vector multiplication operations.

M. Kronbichler and K. Ljungkvist [16] have utilized a matrix-free geometric multigrid approach for solving the Laplace problem with varying polynomial degrees, ranging from moderate to high. The authors conducted a comparative analysis of the performance between an Nvidia Pascal P100

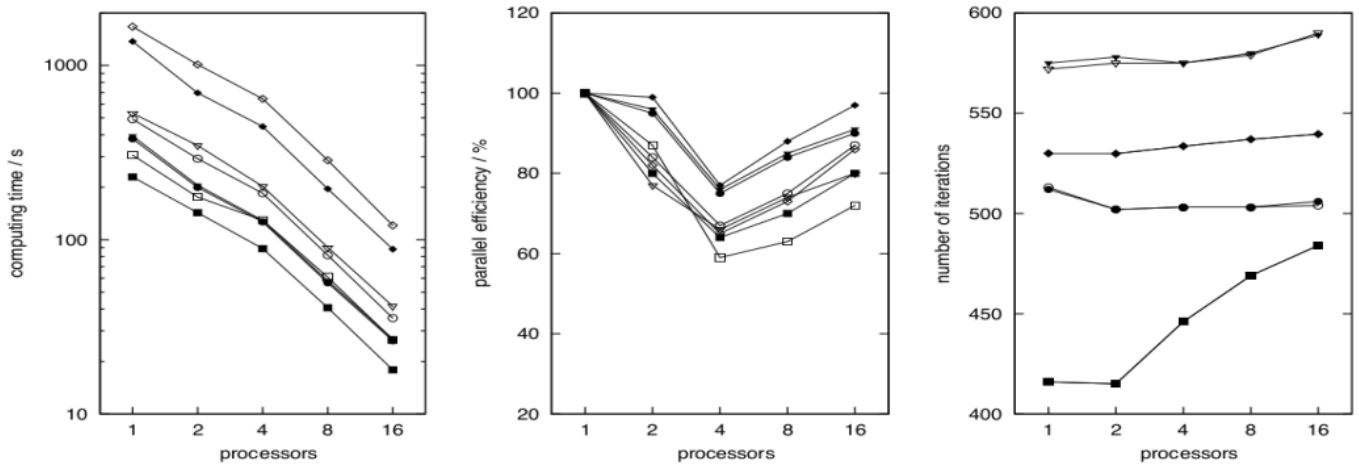

Fig. 4. Runtime of solution phase (left), parallel efficiency of solution phase (middle), and cumulative iteration count (right): □: ams1cg, ◯: amf1cg, ▽: amggs3, ◊: ichpcg (number of iterations divided by 5); empty symbols: double precision, filled symbols: single precision [15]

GPU and Intel Broadwell CPUs, which are renowned for their exceptional multicore execution optimization. Results have revealed that the GPU implementation outperformed the CPU counterpart by a factor of approximately 1.5× to 2× across four different scenarios. Furthermore, their solver infrastructure enables the utilization of mixed precision arithmetic which executed a multigrid V-cycle in FP32 while employing an outer correction in FP64, thereby enhancing throughput by up to 83%.

K. L. Oo and A. Vogel [17] have employed a high precision iterative refinement with geometric multigrid preconditioning in pure FP16 and in FP16-FP32-FP64 combinations across the grid hierarchy for solving large sparse systems of equations that arise from the discretization of 2D Poisson's equation. The experimental results indicate that the iteration count is minimally impacted by the utilization of lower precision. Finally, notable reductions in solver runtime are observed due to decreased memory transfer and an impressive overall speedup of up to 2.5× is achieved for the solver compared to the FP64 baseline.

IV. Error Analysis of Mixed Precision Multigrid Methods

The error analysis of numerical methods holds significant importance in comprehending the intricate balance between computational efficiency and numerical accuracy. Such analysis encompasses the evaluation of discretization errors, interpolation errors, and rounding errors, which principally consists of mathematical investigations. However, only a limited number of scholarly articles have been published in the domain of mixed precision multigrid methods, specifically focusing on error analysis. This section presents a summary based on the work of S. F. McCormick et al. [10], which is the first paper establishing a theoretical framework primarily for rounding error analysis in mixed precision multigrid methods.

A. Iterative Refinement Algorithm

In this and subsequent subsections, $\bar{u} \leq u \leq \dot{u}$ is imposed, where u represents the unit roundoff of a specific precision (u-precision), $\bar{u}$ denotes the unit roundoff of a higher precision ($\bar{u}$-precision), and $\dot{u}$ corresponds to the unit roundoff of a lower precision ($\dot{u}$-precision).

The iterative refinement to solve a sparse semi-positive definite linear system of equations is presented in Algorithm 1. The residual $Ax - b$ is calculated in $\bar{u}$-precision but subsequently rounded to u-precision. Additionally, the inner solver is executed in $\dot{u}$-precision, while all the other operations are carried out in u-precision.

Algorithm 1 Iterative Refinement (IR) [10]

Input A, b, x initial guess, $tol > 0$ convergence tolerance
$r \leftarrow Ax - b$
if $\|r\| < tol$ **then**
return x
end if
$y \leftarrow \text{InnerSolve}(A, r)$
$x \leftarrow x - y$
goto $r \leftarrow Ax - b$

For any $x \in \mathbb{R}^n$, define a discrete energy norm by $\|x\|_A = \left\|A^{\frac{1}{2}}x\right\|$. Additionally, similar to the condition number $\kappa(A) = \|A\| \cdot \left\|A^{-1}\right\|$, $\psi(A) = \||A|\|$ and $\underline{\kappa}(A) = \psi(A)\left\|A^{-1}\right\|$ are defined. The error-related bounds are defined by $\dot{\tau} = \kappa^{\frac{1}{2}}\dot{u}, \tau = \kappa^{\frac{1}{2}}u, \bar{\tau} = \kappa\bar{u}$ and $\gamma = \frac{\kappa^{\frac{1}{2}}+\kappa}{\kappa}$. Finally, a term regarding matrix sparsity is defined by

$$\bar{m}_A^+ = \frac{m_A + 1}{1 - (m_A + 1)\bar{u}}, \tag{8}$$

where m_A is the maximum possible number of nonzeros in each row of the matrix A.

For the inner solver, there exists a $\rho < 1$ such that

$$\left\|y - A^{-1}r\right\|_A \leq \rho \left\|A^{-1}r\right\|_A. \tag{9}$$

Then the relative error bound of the solution is given by

$$\frac{\left\|x^{(i+1)} - A^{-1}b\right\|_A}{\left\|A^{-1}b\right\|_A} \leq \rho_{ir} \frac{\left\|x^{(i)} - A^{-1}b\right\|_A}{\left\|A^{-1}b\right\|_A} + \chi, \tag{10}$$

where the convergence factor is given by

$$\rho_{ir} = \rho + \delta_{\rho_{ir}},\ \delta_{\rho_{ir}} = \frac{(1+2\rho)\tau + \gamma(1+\rho)(1+u)\bar{m}_A^+\bar{\tau}}{1-\tau} \tag{11}$$

and the limiting accuracy is given by

$$\chi = \frac{\tau + \gamma(1+\rho)(1+u)\bar{m}_A^+\bar{\tau}}{1-\tau}. \tag{12}$$

Although the aforementioned equations establish convergence for iterative refinement in the form of energy norm when $\delta_{\rho_{ir}} < 1-\rho$, it will fail when the relative error approaches χ.

B. Two-Grid Algorithm

Algorithm 2 introduces a two-grid correction scheme that serves as the inner solver entirely executed in $\dot{u}$-precision within Algorithm 1. This algorithm involves a relaxation operation denoted by $x \leftarrow x - M(Ax - b)$ where $M \in \mathbb{R}^{n\times n}$ is nonsingular and a prolongation operation represented by $P \in \mathbb{R}^{n\times n_c}$, where n_c is the problem size of the coarse grid. In addition, the scheme utilizes an approximate factor $B_c \in \mathbb{R}^{n_c\times n_c}$, which exhibits slight deviations from the identity matrix I_c on the coarse grid. In the general scenario where $\|B_c - I_c\|_{A_c} < 1$, for any $y \in \mathbb{R}^{n\times n}$ satisfying $\|y\|_A = 1$, it can be demonstrated that the following inequality holds:

$$\left\|(I - PB_c(P^TAP)^{-1}P^TA)Gy\right\|_A^2 < 1. \tag{13}$$

Here $G = I - MA$ is the error propagation matrix.

Algorithm 2 Two-Grid (TG) Correction Scheme [10]

Input A, r, P, M
$r \leftarrow r$
$y \leftarrow Mr$
$r_{tg} \leftarrow Ay - r$
$b_c \leftarrow P^T r_{tg}$
$d_c \leftarrow B_c(P^TAP)^{-1}b_c$
$d \leftarrow Pd_c$
$y \leftarrow y - d$
return y

Furthermore, the authors ultimately establish the convergence of such a two-grid cycle as expressed by the inequality

$$\left\|y - A^{-1}r\right\|_A \leq \rho_{tg} \left\|A^{-1}r\right\|_A, \tag{14}$$

where $\rho_{tg} = \rho_{tg}^* + \delta_{\rho_{tg}}$ and $\delta_{\rho_{tg}}$ is a cubic polynomial in $\dot{\tau}$, a sufficiently small value, such that $\rho_{tg}^* + \delta_{\rho_{tg}} < 1$. As an inner solver, TG consists of a fixed number of elementary arithmetic operations and only involves $\kappa^{\frac{1}{2}}$, thus low precision is sufficient. However, the outer IR involves κ and requires high precision.

C. V-Cycle Multigrid Algorithm

Algorithm 3 demonstrates a V(1,0)-cycle procedure purely in $\dot{u}$-precision. Denote the levels of grids from coarsest to finest as $j \in \{1, 2, ..., l\}$ and let $T_j = I_j - P_j A_{j-1}^{-1} P_j^T A_j$, the error propagation matrices can be defined by

$$V_1 = G_1 \ and\ V_j = (P_j V_{j-1} A_{j-1}^{-1} P_j^T A_j + T_j)G_j,\ 2 \leq j \leq l, \tag{15}$$

and there exists $\rho_v^* \in [0, 1)$ such that

$$\|V_j\|_{A\,j} \leq \rho_v^*,\ 1 \leq j \leq l. \tag{16}$$

Algorithm 3 V(1, 0)-Cycle (V) Correction Scheme [10]

Input $A, r, P, l \geq 1$ V levels
$r \leftarrow r$
$y \leftarrow Mr$
if $l > 1$ **then**
$r_v \leftarrow Ay - r$
$r_{l-1} \leftarrow P^T r_v$
$d_{l-1} \leftarrow$V$(A_{l-1}, r_{l-1}, P_{l-1}, l-1)$
$d \leftarrow Pd_{l-1}$
$y \leftarrow y - d$
end if
return y

By defining the precision coarsening factor $\dot{\zeta}_j = \frac{\dot{u}_{j-1}}{\dot{u}_j}$, $2 \leq j \leq l$ and $\vartheta = min_{1\leq j\leq l}\left\{\theta_j \dot{\zeta}^{-\frac{1}{m}}\right\}$, the authors additionally introduce a progressive precision V-cycle, where the precision is tailored to each level in the grid hierarchy. They establish the convergence of this procedure based on the inequality

$$\left\|y - A^{-1}r\right\|_A \leq \rho_v \left\|A^{-1}r\right\|_A, \tag{17}$$

where $\rho_v = \rho_v^* + \delta_{\rho_v}$ and $\delta_{\rho_v} = \frac{\vartheta^m}{\vartheta^m - 1}\delta_{\rho_{tg}}(\dot{\tau}_l)$. In this equation, $\dot{\tau}_j$ is chosen to be sufficiently small such that $\delta_{\rho_v} = \frac{\vartheta^m}{\vartheta^m-1}\delta_{\rho_{tg}}(\dot{\tau}_l) < 1 - \rho_v^*$ holds for $1 \leq j \leq l$ where l represents the level of the grid. It is noteworthy that in the energy norm, the relative algebraic error exhibits a faster growth rate for fixed precision compared to both mixed precision and progressive precision strategies.

D. Full Multigrid Algorithm

Algorithm 4 introduces a standalone full multigrid procedure aimed to solve a linear system of equations. The nested iterative refinement approach involves performing residual calculations with $\bar{u}$-precision while employing a $\dot{u}$-precision V-cycle as an inner solver within the process. All remaining operations are carried out using u-precision. The selection of an appropriate number of iterative refinement cycles is of crucial importance to ensure convergence within the desired level of discretization accuracy on each grid level.

Algorithm 4 FMG(1,0)-Cycle (FMG) [10]

Input A, b, P, $N \geq 1$ IR cycles (using one V(1, 0)each), $l \geq 1$ FMG levels $x \leftarrow 0$
if $l > 1$ **then**
$x_{l-1} \leftarrow \text{FMG}(A_{l-1}, b_{l-1}, P_{l-1}, l-1, N)$
$x \leftarrow P x_{l-1}$
end if
$i \leftarrow 0$
while $i < N$ **do**
$r \leftarrow Ax - b$
$y \leftarrow \text{V}(A, r, P, l)$
$i \leftarrow i + 1$
$x \leftarrow x - y$
end while
return x

If there exist a sufficiently small χ and a sufficiently large N, the convergence can be established on all $j \in \{1, 2, ..., l\}$ as follows:

$$(\rho_\upsilon + \delta_{\rho_{ir}})^N((\sqrt{2}+\mu\tau)\theta^q C h^q + \mu\tau) + \frac{\chi}{1-(\rho_\upsilon + \delta_{\rho_{ir}})} \leq C h^q, \tag{18}$$

where $\rho_\upsilon + \delta_{\rho_{ir}} < 1$, C and q are positive constants, θ is pseudo mesh-refinement factor, and μ is a function of $\dot{\zeta}_j$. Furthermore, Equation (18) can also be written as

$$(\rho_\upsilon^* + \mathcal{O}(\dot{\tau}))^N(\sqrt{2}\theta^q C h^q + \mathcal{O}(\tau)) + \mathcal{O} \leq C h^q. \tag{19}$$

The full multigrid technique also exhibits the ability to exploit progressive precision, wherein successively lower precision is employed on coarser levels of grid hierarchy. By utilizing this progressive precision strategy, the full multigrid algorithm effectively achieves discretization accuracy while optimizing resource utilization to a minimal extent.

V. Conclusion

After providing an introductory overview of fundamental background knowledge and mathematical concepts, along with a numerical experiment to showcase the potential of mixed precision algorithms, this paper proceeds to examine the current state-of-the-art studies in both practical and theoretical realms. The utilization of mixed precision algorithms has gained significant popularity across various scientific and engineering domains, particularly within the area of machine learning, as a means of enhancing the overall computational efficiency of applications. Notably, considerable attention has been devoted to the integration of purely low precision multigrid techniques into high precision solvers. Nevertheless, the exploration of employing distinct precisions across grid hierarchies to further optimize computational efficiency has been relatively understudied.

Several key areas necessitate further research and development, including the determination of an optimal precision allocation strategy, the establishment of a stability analysis framework, and the advancement of efficient hardware architectures and software frameworks. These aspects demand thorough investigation to overcome the existing limitations and enhance the efficacy of mixed precision algorithms in computational applications.

Acknowledgment

I would like to express my sincere gratitude to Marc Marot-Lassauzaie, my supervisor, for his guidance and support throughout my research. His expertise, feedback, and encouragement have been invaluable in shaping this project.

References

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, vol. 35, pp. 344–369, July 2021.

[2] N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, vol. 31, pp. 347–414, May 2022.

[3] Y. Idomura, T. Ina, Y. Ali, and T. Imamura, "Acceleration of Fusion Plasma Turbulence Simulations using the Mixed-Precision Communication-Avoiding Krylov Method," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, Nov. 2020.

[4] T. N. Palmer, "More reliable forecasts with less precise computations: a fast-track route to cloud-resolved weather and climate simulators?," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 372, p. 20130391, June 2014. Publisher: Royal Society.

[5] T. N. Palmer, "The physics of numerical analysis: a climate modelling case study," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, p. 20190058, Jan. 2020. Publisher: Royal Society.

[6] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," Sept. 2015. arXiv:1412.7024 [cs].

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, (Lille, France), pp. 1737–1746, JMLR.org, July 2015.

[8] F. E. Curtis and K. Scheinberg, "Optimization Methods for Supervised Machine Learning: From Linear Models to Deep Learning," pp. 89–113, Sept. 2017.

[9] W. Hackbusch, *Multi-Grid Methods and Applications*, vol. 4 of *Springer Series in Computational Mathematics*. Berlin, Heidelberg: Springer, 1985.

[10] S. F. McCormick, J. Benzaken, and R. Tamstorf, "Algebraic Error Analysis for Mixed-Precision Multigrid Solvers," *SIAM Journal on Scientific Computing*, vol. 43, pp. S392–S419, Jan. 2021.

[11] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–20, Oct. 1985. Conference Name: ANSI/IEEE Std 754-1985.

[12] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug. 2008. Conference Name: IEEE Std 754-2008.

[13] N. J. Higham, *Accuracy and stability of numerical algorithms*. Philadelphia: Society for Industrial and Applied Mathematics, 2nd ed ed., 2002.

[14] S.-i. O'uchi, H. Fuketa, T. Ikegami, W. Nogami, T. Matsukawa, T. Kudoh, and R. Takano, "Image-Classifier Deep Convolutional Neural Network Training by 9-bit Dedicated Hardware to Realize Validation Accuracy and Energy Efficiency Superior to the Half Precision Floating Point Format," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018. ISSN: 2379-447X.

[15] M. Emans and A. van der Meer, "Mixed-precision AMG as linear equation solver for definite systems," *Procedia Computer Science*, vol. 1, pp. 175–183, May 2010.

[16] M. Kronbichler and K. Ljungkvist, "Multigrid for Matrix-Free High-Order Finite Element Computations on Graphics Processors," *ACM Transactions on Parallel Computing*, vol. 6, pp. 2:1–2:32, May 2019.

[17] K. L. Oo and A. Vogel, "Accelerating Geometric Multigrid Preconditioning with Half-Precision Arithmetic on GPUs," July 2020. arXiv:2007.07539 [cs].