

Hybrid Coarse-Grained Transprecision boosted by a Custom GEMM Kernel to accelerate Parametric Linear Systems

Hamid Noori

Queen's University Belfast

Hans Vandierendonck

Queen's University Belfast

Roger Woods

R.Woods@qub.ac.uk

Queen's University Belfast

Nick Polydorides

University of Edinburgh

Research Article

Keywords: Parametric Linear Systems, Matrix Multiplication, Low/Mixed-Precision, Transprecision Computing, Custom GEMM Kernel

Posted Date: October 29th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-8730840/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Hybrid Coarse-Grained Transprecision boosted by a Custom GEMM Kernel to accelerate Parametric Linear Systems

Hamid Noori¹, Hans Vandierendonck¹, Roger Woods^{1*},
Nick Polydorides²

^{1*}School of Electronics, Electrical Engineering and Computer Science,
Queen's University Belfast, Belfast, UK.

²Institute for Data, Imaging & Comms., University of Edinburgh,
Edinburgh, UK.

*Corresponding author(s). E-mail(s): r.woods@qub.ac.uk;
Contributing authors: h.noori@qub.ac.uk; h.vandierendonck@qub.ac.uk;
n.polydorides@ed.ac.uk;

Abstract

Processing parametric linear systems of the form $G(P) := X^T P X$ is computationally expensive due to the large size of matrix X and reliance on double-precision floating-point arithmetic. As many applications can tolerate small numerical errors, performance trade-offs can achieve low-precision format GPU solutions. A hybrid coarse-grained transprecision computing framework is proposed that comprises an offline phase that identifies suitable low-precision formats for matrices X and P under a predefined error constraint, and an online phase that dynamically analyzes each incoming P to select the optimal precision for computing $G(P)$. It allows a $21.6M \times 500$ matrix, X , to be processed up to 58x faster than its double-precision version, enabling real-time evaluation of 2 P matrices/s on an NVIDIA A100 GPU, with an average relative error below 0.6%. In addition, a fused custom GEMM kernel for computing $G(P) := X^T P X$ has also been developed, by integrating two standard CUDA-based BLAS operations into a single CUTLASS-based kernel. It eliminates intermediate memory transfers and reduces computational overhead, providing an additional 3x speedup over the conventional implementation.

Keywords: Parametric Linear Systems, Matrix Multiplication, Low/Mixed-Precision, Transprecision Computing, Custom GEMM Kernel

1 Introduction

Numerical approximation methods for Partial Differential Equations (PDEs), such as the finite element (FE) methods, have been the cornerstone of scientific computing. They provide a principled and versatile framework for modelling physical phenomena and engineering systems at realistic levels of detail. By transitioning from the calculus of continuous functions to matrix-based formulations, they enable robust analysis through algebraic systems whose solvability and error properties are well studied. As a result, PDE-based simulations have become indispensable in fields such as structural mechanics, fluid dynamics, electromagnetics, and material science [1].

Their computational challenges are particularly pronounced as the discretisation of complex geometries or high-accuracy requirements leads to systems with tens or hundreds of millions of degrees of freedom. While high-performance computing infrastructures such as GPU clusters can handle large-scale simulations offline, the landscape shifts dramatically when simulation is required in real time. Applications such as digital manufacturing twins, robotics and diagnostic monitoring require predictive models that operate within strict latency and energy constraints, often on embedded or edge hardware where computational resources, memory capacity, and power budgets are limited. In these settings, exhaustive high-fidelity solutions are infeasible and efficient formulations that balance accuracy with computational speed are essential [2].

A central computational bottleneck arises in the many-query paradigm, where repeated solutions of similar systems are required for varying parameters. In the FE method, for example, the stiffness matrix must be reassembled for each new set of material or geometric parameters, while in parametric statistical or signal processing tasks, repeated evaluation of weighted quadratic forms is required. This scenario can be abstracted into the study of parametric linear systems of the form,

$$G(P)u = b, \quad \text{where} \quad G(P) = X^\top P X, \quad (1)$$

where X is a dense tall-and-skinny matrix, P is a diagonal matrix of positive entries defined over the rows of X and u, b are vectors. Despite $G(P)$ often being low-dimensional, its construction is expensive since X may be extremely large due to the scale of discretisation/number of measurements. This structure arises naturally in multiple domains: in FE analysis and structural engineering, where P encodes varying material properties; in graph theory, where X is an incidence matrix and P represents edge weights and; in signal processing optimization where $G(P)$ can represent Hessians or spectral weighting matrices [1]. The challenge lies not only in computing $G(P)$ efficiently, but in doing so repeatedly as new P instances arrive.

The significant computational cost associated with constructing $G(P)$ necessitates the creation of highly efficient methodologies, particularly for real-time scenarios involving numerous queries of the form (X, P) , where the matrix X is fixed and the parameter matrix P is continuously updated. A primary objective here is to enhance the throughput at which successive P matrices can be processed. The solution of parametric linear systems often exhibits a natural tolerance to numerical inaccuracies, opening the opportunity to exploit lower-precision arithmetic as variables and parameters typically do not require standard double-precision floating-point (FP64)

format. Modern GPUs provide specialized hardware support such as NVIDIA Tensor and AMD Matrix cores, that delivers substantially higher performance and energy efficiency for lower-precision computations, e.g., standard half precision floating-point (FP16), 16-bit floating-point (BF16). Reduced precision also lowers the memory footprint as compact formats can enable the problem to fit entirely within the limited GPU memory, thereby avoiding costly performance penalties.

Transprecision computing has now emerged as a compelling paradigm to trade precision for performance within application-specific error tolerances [3]. Numerous precision-tuning techniques operate within traditional formats such as FP64, single precision floating-point (FP32), or FP16, but frequently lack support for modern low-precision formats such as TensorFloat-32 (TF32) and BF16 [4–14]. Furthermore, existing approaches face two critical limitations: some fail to meet essential accuracy requirements such as an error threshold below 1% [14, 15], whilst others incur significant runtime overhead, rendering them impractical for real-time applications [6].

The diagonal structure of the P matrices further motivates the design of a custom, fused GEMM (General Matrix Multiplication) kernel, where tiles of X are scaled by the corresponding P elements while being loaded from global memory and stored into shared memory. It offers key advantages over the two-step method of first scaling X and then invoking GEMM to compute $G(P)$. Firstly, it eliminates the need for a preprocessing kernel that scales each column of X , saving one full pass over global memory and reducing bandwidth. Secondly, by integrating scaling directly into the GEMM main loop, it can improve cache reuse while avoiding extra synchronization and kernel-launch overhead. Finally, the fused method reduces memory footprint by eliminating the intermediate scaled copy of X ($PX = P \times X$). Collectively, these factors motivate the development of mixed-precision and lower-precision solvers accelerated by a custom GEMM kernel for parametric linear systems.

This paper presents two orthogonal approaches for accelerating the solution of parametric linear systems, a hybrid coarse-grained transprecision computing framework, and a fused custom GEMM kernel. The proposed hybrid transprecision methodology dynamically selects suitable low-precision arithmetic formats on NVIDIA GPUs for computing the G matrix. An *offline* analysis phase profiles a representative subset of P to identify an appropriate set of low-precision formats under a predefined accuracy constraint. An *online* execution phase dynamically selects the optimal format, based on the data range of the incoming P matrices. In addition, a fused custom GEMM kernel is introduced, which merges the two conventional operations—($P \times X$) and ($G = X^\top \times (P \times X)$)—into a single CUTLASS-based GEMM operation, removing intermediate memory transfers and redundant computations and yielding a 3x performance improvement. The CUTLASS GEMM configuration was carefully tuned for the target data type and GPU architecture to achieve optimal efficiency. Furthermore, a *Recompute* strategy is proposed for very large X and PX matrices that do not fit in GPU memory. This method minimizes or avoids out-of-core (OOC) computation at the expense of slightly higher numerical error. Overall, the proposed approaches aim to maximize performance and energy efficiency while maintaining solution accuracy within a user-defined error threshold.

Our key contributions are as follows:

- A hybrid, coarse-grained transprecision methodology that explores lower-precision data types (FP32, TF32, BF16, and FP16) in an offline phase and uses a lightweight online phase that makes it suitable for real-time processing of parametric systems.
- A comprehensive cross-architectural analysis of the trade-offs between accuracy, performance, and energy efficiency across three generations of NVIDIA GPU architectures: Hopper (H100), Ampere (A100, GeForce RTX 3070), and Volta (V100). It achieves a maximum speedup of 117x, reduces energy consumption by 36.6x, and maintains an average relative error of only 0.51%.
- A memory-efficient computation strategy that combines lower-precision arithmetic with optimized data reuse, thereby reducing GPU memory consumption by 75%, thereby avoiding or minimizing the need for OOC computations and associated performance penalties.
- A fused CUTLASS-based custom GEMM kernel tailored for parametric linear systems that eliminates redundant memory operations and enhances computational efficiency, achieving performance improvements of up to $\sim 3x$.

This work extends our earlier publication [16] through a more detailed description of the approach, the development of fused custom GEMM kernel and its detailed performance evaluation, a new *Recompute* strategy to replace the *Reload* approach (allowing handling of large matrices that exceed GPU memory capacity), additional energy consumption and power savings analysis results and extension of the analysis to the Hopper architecture.

The remainder of this paper is organized as follows. Section 3 reviews related work. Section 2 outlines the motivation and the problem statement. The proposed three-layered framework including the hybrid coarse-grained transprecision methodology, the *Recompute* strategy, and design of the fused custom GEMM kernel, are detailed in Section 4. Section 5 presents and discusses the experimental results, and Section 6 provides concluding remarks.

2 Motivation and Problem Statement

Previous work [17], [18], [19] on optimizing irregularly shaped matrices using conventional GEMM techniques has targeted regular, square-shaped inputs and does not apply to tall-and-skinny matrices, such as those in Table 1. In this case, algorithms are proposed to match the performance characteristics and memory access patterns of tall-and-skinny matrix operations on GPUs.

The matrices under investigation originate from representative parametric linear systems exploited in electrostatic simulations using the FE method. The X matrix arises from the projection of the discretized gradient operator onto linear tetrahedral elements over a meshed spherical domain. The P diagonal matrix encodes a spatially varying material property, such as electrical conductivity in electrostatic simulations or doping profiles in semiconductor device modeling. The resulting system, $G(P) = X^T P X$, captures the weighted interactions of these properties across the mesh, and solving $G(P)u = b$ provides a numerical approximation of the electrostatic potential distribution.

The P value distribution spans from 10^{-3} to 10^{+3} , yielding a maximum-to-minimum ratio of 10^5 . The data ranges confirm that X does not fully exploit the dynamic range provided by FP64 floating point. Due to their large sizes, these matrices involve a substantial number of multiplications and additions, as well as significant memory storage. The matrix, $Q = X^\top X$, is defined as $G(P)$ where P is the identity diagonal matrix. Table 1 gives the dimensions and sizes of the X and Q for seven different parametric linear systems, sorted from smallest to largest with Table 2 reporting maximum and minimum absolute values [16]. It can be seen that X and Q matrices utilize only a fraction of the dynamic range available. Since G is small, u is computed using double. For double precision, the bigger the range in the values of P , the higher the condition number of G will be.

Table 1 Size of matrices

Matrices	X Dimensions	size of X (GB)	G/Q Dimensions
M1	2053680 * 50	0.76	50 * 50
M2	5066607 * 50	1.9	50 * 50
M3	21567126 * 50	8.03	50 * 50
M4	5066607 * 200	7.6	200 * 200
M5	5066607 * 500	18.9	500 * 500
M6	21567126 * 200	32.2	200 * 200
M7	21567126 * 500	80.3	500 * 500

As modern GPUs support a wider range of arithmetic formats, FP32, TF32, BF16, and FP16, it is promising to use this capability to improve performance by computing G using a lower-precision format whilst maintaining a relative error below 1% compared to the reference solution. Employing lower-precision formats both accelerates the computation and significantly reduces energy consumption. Moreover, the diagonal structure of the parameter matrices P presents a key optimization opportunity of fusing the scaling of X directly into the GEMM operation. Our custom kernel realizes this by scaling tiles of X with corresponding elements of P during the load from global to shared memory. This yields three critical performance advantages over the conventional two-step approach by: (1) obviating a separate preprocessing kernel, eliminating a full global memory pass and reducing bandwidth pressure; (2) enhancing cache locality and avoiding the latency of extra synchronizations and kernel launches; and (3) reducing the overall memory footprint by avoiding the storage of an intermediate scaled matrix $P \times X$. Based on this motivation, this work addresses the following problem statements:

- Can lower-precision arithmetic be reliably used for large-scale parametric linear systems while adhering to a strict (e.g., 1%) error tolerance?
- How can we determine the appropriate lower-precision format for a given P matrix to guarantee this error bound is met?
- What are the precise trade-offs between computational speed, energy efficiency, and numerical accuracy when employing such transprecision computing?

- To what extent does the fused custom GEMM kernel improve performance and energy efficiency compared to a conventional two-step BLAS approach?

Table 2 Range of data of matrices values

Matrices	X Abs Max	X Abs Min	Q Abs Max	Q Abs Min
M1	0.228	6.4552e-14	1.5344	1.9963e-08
M2	17.188	6.8888e-12	8201.3	3.2388e-04
M3	24.303	3.6469e-13	9315.0	1.7253e-05
M4	0.129	8.5878e-12	1535.1	9.8658e-05
M5	0.241	8.5878e-12	2699.4	6.0961e-05
M6	0.071	5.6628e-13	1498.9	5.8712e-05
M7	0.122	5.6628e-13	2593.8	6.0604e-06

3 Related Work

The related work addresses both tall-and-skinny matrix-matrix multiplication on GPUs and precision customization.

Chen et al. [17] explored optimization techniques for GEMM operations involving one large and another tall-and-skinny matrix. Subsequent research [20] extended this to scenarios where a tall-and-skinny matrix is multiplied by a smaller, regular-shaped matrix. It emphasized maximizing off-chip memory efficiency, utilizing shared memory, and overlapping computation with memory access latency. Tang et al. [18] explores mixed-precision GEMM implementations for two main cases on GPUs, namely computing $A^T B$, where both A and B are tall-and-skinny, and computing AB , where A is tall-and-skinny and B is a small square matrix. Performance improvements are achieved through optimized task scheduling, refined memory access patterns, and exploitation of Tensor Core capabilities. Ernst et al. [19] focuses on high-performance double-precision computations for both real and complex tall-and-skinny matrices. Targeting CUDA, they leverage a code generation framework that explores diverse thread mapping and tiling configurations to identify optimal performance settings.

With an increasing focus on precision customization, various tools and methodologies have been developed to assist developers in adapting applications originally implemented with high-precision floating-point formats to more efficient lower-precision alternatives. The annotation-based framework in [4] allows programmers to specify which floating-point types should be replaced by fixed-point. Rojek [6] employs a modified random forest algorithm to explore a wide configuration space, systematically pruning suboptimal configurations to identify the most effective setup, but incurs significant runtime overhead. GPU-FPtuner [12] is a mixed-precision autotuning framework designed for floating-point applications on GPUs, which progressively reduces variable precision using a combination of static analysis and dynamic profiling until either all variables reach single precision or a predefined error bound is met.

It employs compiler-assisted analysis, while considering GPU-specific aspects such as kernel configuration, shared memory utilization, and atomic operations.

GRAM [7] introduces an approach to mixed-precision tuning on GPUs by assigning precision levels to data elements rather than variables, formulating precision tuning as a data-wise approximation problem and is viewed as a soft variant of loop perforation. Given a CUDA program and accuracy target, it improves performance by combining FP16–FP32 or FP32–FP64 arithmetic, but not FP32 and BF16 and cannot be applied to highly optimized closed-source libraries such as cuBLAS or cuBLASLt. POP [8] employs a static, constraint-based precision tuning approach that models numerical error propagation and accuracy needs using semantic equations, to determine optimal bit-level precisions. Its integer linear programming optimization allows theoretical identification of optimal precisions independent of predefined data types, but its static, offline nature limits reaction to dynamic workloads and runtime operation.

Ren et al. [10] propose an autotuning strategy using the “Blocks” directive in the ppOpen-AT autotuning framework [21], allowing developers to selectively assign lower precisions to specific code regions to balance performance and accuracy. It is designed for CPU-based applications, supporting only double and single precision. PrecTuner [11] introduces a holistic, mixed-precision, auto-tuning framework that identifies affine numerical kernels and applies loop transformations to generate mixed-precision variants, optimizing performance under a user-defined accuracy constraint. Its innovation lies in splitting loop dimensions to balance reduced-precision computation and accuracy loss, but only supports FP32 and FP64 formats and targets open-source CUDA kernels and not closed-source, highly optimized GPU libraries such as cuBLAS or cuBLASLt. MixPert[9] introduces, a system that balances accuracy and performance by optimizing FP32 emulation on GPU Integer Tensor Cores.

These approaches lack support for commonly used low-precision formats such as FP32, TF32, FP16, and BF16, and are limited to closed-source, highly optimized libraries like cuBLAS. In contrast, our method adopts a coarse-grained strategy, selecting appropriate low-precision formats at a higher level, which allows integration with black-box libraries and broader applicability. A survey of precision customization tools [22] concludes that although several mature solutions have emerged, significant gaps remain—particularly in static-analysis-based approaches and in the integration of precision tuning techniques into mainstream, industry-grade compiler frameworks.

3.1 Contribution

Whilst the work in [17], [18] and [19] assumes fixed precision formats and primarily optimizes execution efficiency, our work differs by (1) dynamically selecting low-precision formats for tall-and-skinny matrices under user-defined error constraints, and (2) introducing a fused custom GEMM kernel. We target GPU architectures and extends mixed-precision tuning to modern low-precision formats—including FP64, FP32, TF32, FP16, and BF16 formats, not covered by existing work. Moreover, we perform dynamic precision selection during an online phase, adapting precision based on the numerical properties of the P matrices. This enables adaptive, data-driven control of precision while exploiting GPU Tensor core acceleration for performance gains.

Unlike [17], we target cases where both operand matrices are tall-and-skinny, although these prior optimization strategies are complementary to our transprecision framework and could be integrated. Moreover, unlike fixed-point conversion methods where the target variables are predetermined, our framework operates dynamically where the P matrices are randomly generated and not known a priori. Our work focuses on enhancing the real-time solution of parametric linear systems, thereby avoiding the significant runtime overhead of [6] and [12].

4 Proposed Orthogonal Approaches

The proposed transprecision methodology accelerates the solution of parametric linear systems while maintaining the solution error within a user-defined accuracy threshold. It dynamically selects the most suitable low-precision arithmetic formats on NVIDIA GPUs for computing the G matrix, through an offline analysis phase, where a representative subset of P matrices is profiled to determine an appropriate candidate low-precision format, and an online execution phase, where the optimal precision format is dynamically selected based on the data range of incoming P matrices. Five alternative solutions are explored and evaluated for computing the G matrix, by utilizing the determined precision formats.

Three implementation strategies were considered: a conventional two-step approach using CUDA-based BLAS routines; symmetric rank-k (SYRK) update routines to compute only the upper or lower triangular part of G , followed by a simple copy to fill the other half and; a fused custom GEMM kernel, which integrates the two traditional kernels ($(P \times X)$ and $(G = X^\top \times (P \times X))$) into a single CUTLASS-based GEMM operation. The fused custom GEMM kernel eliminates intermediate memory transfers and redundant computations, significantly enhancing both performance and energy efficiency, but needs to be tuned for each target data type and GPU architecture.

4.1 Hybrid Coarse-Grained Transprecision Methodology

The target systems present several challenges: stringent real-time constraints; strict accuracy requirements (error $< 1\%$) and; extremely high computational complexity where computing a single value may require up to 42 million multiplications and additions. Moreover, balancing precision across multiple low-precision formats (FP32, TF32, BF16, and FP16) further complicates the process. Therefore, a hybrid coarse-grained transprecision computing methodology (Fig. 1) comprising an offline and online phase, has been developed to address these challenges.

Offline Phase: This phase (illustrated in Algorithm 1) evaluates whether a candidate low-precision (CLP) format, combined with or without normalization, can deliver results within the required accuracy threshold. The process begins with the most compact precision format and progresses to higher precisions if needed. Reference computations are performed using FP64 precision, following Equations (2)–(3), to establish baseline accuracy. Specifically, Equation (2) calculates G with the identity P matrix (which we call it Q), and Equation (3) solves for the reference u vector used for error comparison.

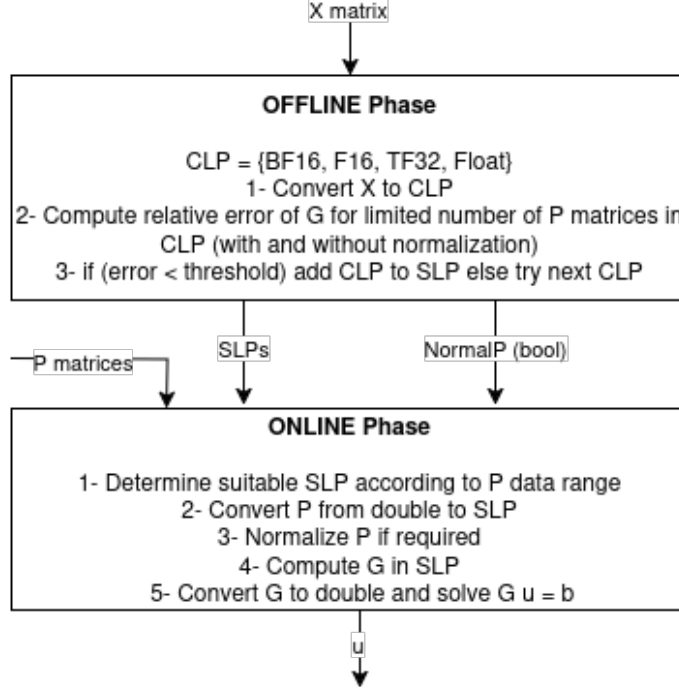


Fig. 1 General overview of the hybrid coarse-grained transprecision approach [16]

$$Q_{\text{FP64}} = X_{\text{FP64}}^{\top} X_{\text{FP64}} \quad (2)$$

$$Q_{\text{FP64}} u_{\text{FP64}} = b \quad (3)$$

where u_{FP64} and b are $s \times 1$ FP64 vectors, X is $m \times s$ and G and Q are $s \times s$ matrices.

The reference solution u_{FP64} obtained from Eq. (3) serves as the baseline for evaluating the accuracy of lower-precision solutions u_{BF16} , u_{FP16} , u_{TF32} , and u_{FP32} . The procedure begins by testing the most compact formats—BF16 and FP16—to identify which yields lower error. The matrix X_{FP64} is converted into X_{BF16} and X_{FP16} , used to compute Q_{BF16} and Q_{FP16} , respectively. These Q matrices are then cast back to FP64 precision to solve Equation (3), generating u_{BF16} and u_{FP16} . Finally, the normalized relative error between each low-precision solution and the FP64-precision reference is computed, where $\text{LP} \in \{\text{BF16}, \text{FP16}\}$ (in general $\text{LP} \in \{\text{BF16}, \text{FP16}, \text{TF32}, \text{FP32}\}$).

$$\text{error}_{\text{LP}} = \frac{\|u_{\text{FP64}} - u_{\text{LP}}\|_2}{\|u_{\text{FP64}}\|_2} \quad (4)$$

The format yielding the smallest error is selected for subsequent evaluation. The matrix X is then cast to the CLP format, and 200 random P matrices are generated across two ranges: a narrow range, P_n , with values range from 1 to 10^{+3} and a wide range, P_w , with values range from 10^{-3} to 100, each constituting 50% of the samples. Since P_n contains larger values, it increases the risk of overflow when computing G , potentially requiring higher precision. Conversely, P_w includes smaller values, which lower the

risk of overflow and allow for reduced precision. For each P range, the corresponding selected low-precision sets, SLP_{Pn} and SLP_{Pw} , are initialized as empty. For each P sample, G is computed in the CLP format, followed by solving for u in FP64 precision and evaluating its relative error using Equation (4). If the error satisfies the threshold, the CLP is added to the corresponding SLP set. Otherwise, P matrices are normalized to enhance numerical stability by mitigating overflow. Normalization is performed by dividing each element of P by the absolute value of the largest element of P . This introduces negligible runtime overhead since the diagonal P matrices are stored as vectors. If the acceptable error is not achieved, the procedure iterates with higher-precision formats—TF32, FP32, and finally FP64. The final selected precision set (SLP_{Pn} , SLP_{Pw}) determine the precision in which X must be stored. When both use FP32 and TF32, a single FP32 version of X suffices, as TF32 operations utilize FP32 input. However, if one uses FP32/TF32 and the other FP16, X must be stored in both formats, increasing GPU memory consumption by approximately 25–50% compared to using a single format, yet still more memory-efficient than FP64 precision.

Online Phase: During the online phase, the precision format for processing incoming P matrices (in FP64 precision) is determined. If SLP_{Pn} and SLP_{Pw} differ, the data range of each P is first examined to select the appropriate low-precision arithmetic. When both are identical, a single SLP format is used, and range checking is unnecessary. The selected P is then converted to the corresponding SLP format and normalized if its normalization flag is set to true.

The matrix G is computed in the chosen SLP format and subsequently cast to FP64 for solving u . Before conversion, any $G[i, j]$ entries equal to $\pm\infty$ are replaced with the largest finite value representable in that format, as very small values would cause instability in computing G^{-1} . Since rounding errors in low-precision arithmetic can cause G to lose symmetry, a symmetrization step is applied to restore it. This additional computation is minimal as G is typically a small (50–500) square matrix.

$$G = \frac{G + G^\top}{2} \quad (5)$$

4.2 Computing G given Memory Constraints

There are several approaches to compute $G = X^T P X$. The *Direct* approach computes PX , and then evaluates $G = X^T(PX)$ whereas the *SquareRootP* approach computes the square root of P , denoted as $P^{\frac{1}{2}}$, and forms $E = P^{\frac{1}{2}}X$, followed by $G = E^T E$. The advantage of *Direct* over *SquareRootP* is that it avoids the overhead of computing $P^{\frac{1}{2}}$. In our case, this overhead is negligible since P is a vector. The strength of *SquareRootP* is that G is a Gram matrix and thus symmetric, enabling the use of efficient symmetric rank-k (SYRK) update routines to compute only the upper or lower triangular part of G , followed by a simple copy to fill the other half. While this reduces computation, these routines often require tuning for tall-and-skinny matrices, or alternatively, may need customized implementations. In particular, we found the cublas[DS]syrr routines to be more efficient than cublas[DS]gemm on square matrices, but they performed poorly on the tall and skinny matrices that are the subject of this work.

Algorithm 1 Offline Phase of the Proposed Approach

Compute Reference Values in FP64 Precision
1: Compute u_{FP64} (Equations (2) - (3))
Select Lower-Precision Format: BF16 vs. FP16
2: Compute $\text{error}_{\text{BF16}}$ (in BF16, Equations (2), (3) and (4))
3: Compute $\text{error}_{\text{FP16}}$ (in FP16, Equations (2), (3) and (4))
4: **if** $\min(\text{error}_{\text{BF16}}, \text{error}_{\text{FP16}}) < \text{errorThreshold}$ **then**
5: $CLP \leftarrow \text{error}_{\text{BF16}} < \text{error}_{\text{FP16}} ? \text{BF16} : \text{FP16}$
6: **else**
7: $CLP \leftarrow \text{TF32}$
8: **end if**
Test Random P Matrices
9: $\text{SLP}_{\text{Pn}, \text{Pw}} = \emptyset$
10: **Label:** GenerateP
11: Generate 200 random P matrices
12: Compute $G(P_{\text{n/w}})_{CLP}$ and convert it to FP64
13: Compute u_{CLP} in FP64 and compute error_{CLP}
14: **if** $\text{error}_{CLP} < \text{errorThreshold}$ **then**
15: $\text{SLP}_{\text{Pn/w}} \leftarrow \text{SLP}_{\text{Pn/w}} \cup CLP$
16: NormalizingP $\leftarrow \text{FALSE}$
17: **return**
18: **end if**
19: **if** $((\text{SLP}_{\text{Pn}} == \emptyset) \text{ or } (\text{SLP}_{\text{Pw}} == \emptyset))$ **then**
20: Normalize P matrices: $P_{\text{norm}} = P / \max(P)$
21: Generate 200 normalized random P matrices
22: Compute $G(P_{\text{n/w}})_{CLP}$ and convert it to FP64
23: Compute u_{CLP} in FP64 and compute error_{CLP}
24: **end if**
25: **if** $\text{error}_{CLP} < \text{errorThreshold}$ **then**
26: $\text{SLP}_{\text{Pn/w}} \leftarrow \text{SLP}_{\text{Pn/w}} \cup CLP$
27: NormalizingP $\leftarrow \text{TRUE}$
28: **return**
29: **end if**
30: Set $CLP \leftarrow \text{NextHigherPrecision}(CLP)$
31: **if** $((\text{SLP}_{\text{Pn}} \neq \emptyset) \text{ and } (\text{SLP}_{\text{Pw}} \neq \emptyset)) \text{ or } (CLP == \text{FP64})$ **then**
32: **if** $(\text{SLP}_{\text{Pn/w}} == \emptyset)$ **then**
33: $\text{SLP}_{\text{Pn/w}} \leftarrow \text{SLP}_{\text{Pn/w}} \cup CLP$
34: **end if**
35: **return**
36: **else**
37: **goto** GenerateP
38: **end if**

Table 3 Comparing Solutions for Computing G

Solution	Computation	Kernels	Storage	Notes
Direct	$Q = X^T(PX)$	GEMV; GEMM	$2 \times \text{size}(X)$	Potential OOC
SquareRootP	$E = P^{1/2}X$; $Q = E^T E$	sqrt; GEMV; SYRK	$2 \times \text{size}(X)$	Potential OOC
Reload	$X = P^{1/2}X$; $Q = X^T X$	sqrt; GEMV; SYRK	$\text{size}(X)$	Transfer X for each P
Recompute	$X = P^{1/2}X$; $Q = X^T X$; $X = X/P^{1/2}$	sqrt; 2×GEMV; SYRK	$\text{size}(X)$	Loss of precision
CustomGEMM	$Q = X^T P X$	Custom kernel	$\text{size}(X)$	Kernel development

Both *Direct* and *SquareRootP* share the key drawback of requiring the storage of both X and PX . Since these matrices are very large here, this can exceed GPU memory capacity and force prohibitively expensive OOC computation. A third option is to compute $P^{1/2}$ but overwrite X in-place with E , thereby halving memory usage and potentially avoiding OOC execution (*Reload* and *Recompute* approaches). The downsides are twofold: the cost of computing $P^{1/2}$ remains, and because X is overwritten, it must be restored for each new P , by either reloading from host memory (*Reload*), which can be time consuming for very large X , or by recomputing and overwriting $X = X/P^{1/2}$ (*Recompute*). Although *Recompute* introduces extra operations, it is faster and more energy-efficient than *Reload* when the data size exceeds a certain threshold, which is hardware- and precision-dependent. However, when X is stored in a low-precision format, *Recompute* can accumulate errors due to repeated arithmetic and recomputation, avoided by *Reload*, albeit at the cost of higher transfer overhead. The advantage of these overwrite-based approaches are that only one copy of X is maintained, memory usage is reduced by 2×, and symmetric rank- k updates can still be leveraged for improved performance—provided they are well-optimized for tall-and-skinny configurations.

The *CustomGEMM* approach involves the design of a fused custom GEMM kernel in which X is scaled by P as X is fetched from global memory before being written into shared memory. It provides key performance benefits over the two-step approach. Firstly, it removes the need for a separate preprocessing kernel, thereby eliminating an additional global memory pass and alleviating memory bandwidth pressure. Secondly, it improves cache locality while avoiding the synchronization delays and kernel-launch overheads associated with multiple kernels. Finally, it reduces the overall memory footprint by eliminating the requirement to store an intermediate scaled matrix PX . The solutions have been summarized in Table 3.

4.3 Fused Custom GEMM Kernel

The CUTLASS library accelerates GEMM by decomposing the computation into a hierarchy of threadblock tiles, warp tiles, and instruction tiles as illustrated in Fig. 2. The output matrix is divided into threadblock tiles, each of which loads portions of matrices A and B from global memory into shared memory. Within each threadblock, warps process smaller warp tiles by fetching data into the register file, while MMA

(Matrix Multiply-Accumulate) instructions operate on the smallest instruction tiles. These are low-level PTX (Parallel Thread Execution) instructions that perform fused matrix-multiply-and-accumulate operations on tensor Cores. This hierarchical tiling maximizes data reuse and maps computation efficiently onto GPU hardware.

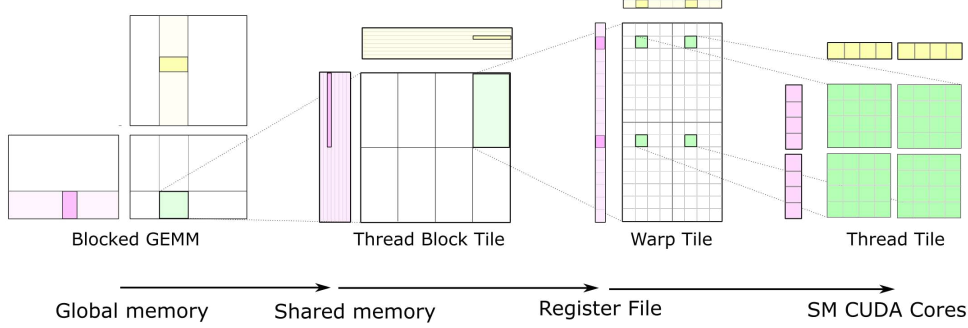


Fig. 2 Overview of the CUTLASS tile hierarchy[23]

To further improve efficiency, CUTLASS employs software pipelining, which overlaps data loading with computation to hide memory latency. GEMM performance is therefore highly sensitive to the choice of tile shapes including threadblock, warp and instruction, as these determine shared memory usage, register pressure, and streaming multiprocessor utilization. These along with the number of pipeline stages, elements per vectorized access, and Split-K slices, are among the key tunable parameters in CUTLASS GEMM. Careful tuning is crucial for achieving high performance across different GPU architectures. Here, it is performed manually.

The *CustomGEMM* approach is a CUTLASS-based kernel, which is developed by modifying NVIDIA’s CUTLASS library so that a single `GemmSplitKParallel` kernel directly computes $G = X^T P X$ without requiring prior computation of PX . In other words, instead of using two BLAS calls for each of the matrix multiplications that define G , we create one customized GPU kernel. By applying column-wise scaling by the diagonal vector P while loading operand B of `GemmSplitKParallel` (or multiplier, in our case it is X) from global memory into shared memory, the diagonal scaling can be fused with the matrix multiplication. To enable this, we extended the `Arguments` and `Params` structures of `"gemm_splitk_parallel.h"` header file to include a device pointer `p_ptr` to the diagonal scaling vector or P matrix which is passed down through the kernel stack into `MmaMultistage`, where it is stored along with `base_B_ptr_` (the base pointer of X) and `problem_size_n_` (the number of columns of X). At the point where operand tiles are loaded in `MmaMultistage::copy_tiles_and_advance()`, we replaced the original asynchronous `cp.async` copy of B (X matrix) with a synchronous load/store loop. Each vector of elements from global memory is unpacked, and for every scalar element, we compute its column index, fetch the corresponding scale factor from P , and multiply it directly before writing the scaled result into shared memory.

This ensures that all subsequent stages of CUTLASS GEMM operate transparently on the already-scaled tiles of B operand (which is X).

The modifications preserve CUTLASS’s programming model: the host code only needs to pass `d_P` (a pointer to the diagonal scaling vector or P matrix) when constructing the Arguments object. Functionally, this integrates scaling into the GEMM kernel itself and trades off some overlap of memory operations and compute, because the synchronous load replaces the optimized `cp.async` path. Nevertheless, for many practical problem sizes the cost is modest, and the benefit is a much simpler fused kernel. Importantly, by storing P as a device pointer in Params, our approach scales to arbitrarily large problems without requiring special layouts or preprocessing of X . Overall, this modification demonstrates a straightforward but powerful extension of CUTLASS to support fused linear algebra kernels of the form $G = X^\top P X$, which appear in statistics, optimization, and machine learning, while preserving the usability of CUTLASS’s device-level GEMM interface.

A key technical detail is the computation of the indices of X and P within the modified `copy_tiles_and_advance()` routine. Since CUTLASS uses vectorized loads (e.g., `cutlass::Array<ElementB, kElements>`), each element must be indexed relative to its position in the original row-major layout of B. To accomplish this, we compute a linear index as the offset of the current global pointer from `base_B_ptr_`, multiplied by the vector length, plus the element index within the vector. The column index is then derived as $col = linear_idx \bmod problem_size_n_$, where `problem_size_n_` is the number of columns of X stored in the GEMM parameters. This `col` value is used to access `p_ptr_[col]`, the diagonal scaling element corresponding to that column. Each loaded element `global_data[t]` is then multiplied by the scale value before being committed to shared memory. This indexing ensures that scaling is applied consistently across all tiles and stages, independent of threadblock shapes or warp-level iterators, and maintains full generality for arbitrary problem sizes. The fused custom GEMM kernel is detailed in Algorithm 2.

We also implemented a version of the fused custom GEMM kernel that leverages asynchronous memory copy. In this approach, using `cp.async` data is transferred directly from global memory to shared memory through a hardware pipeline, bypassing registers. However, this mechanism requires memory fences to ensure that all `cp.async` operations for a group of vectors have completed before the data can be safely accessed in shared memory. Only after synchronization can the vectors be scaled in-place within shared memory. Due to these synchronization overheads and pipeline constraints, the fused custom GEMM kernel with asynchronous copy achieves approximately 37% lower performance compared to its synchronous copy counterpart.

5 Experimental Setup and Results

The evaluation was performed on the seven matrix sets (Table 1), using 10,000 randomly generated parameter matrices P (which is 100x of number of P matrices used during the offline phase) across four NVIDIA GPUs from three different generations, representing significant evolutions in specialized hardware. For the proposed approaches, the average speedup, relative error, energy and power savings are reported

Algorithm 2 Fused GEMM Kernel for $G = X^T P X$

Require: Input matrices $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{M \times K}$, diagonal scaling vector $P \in \mathbb{R}^K$

Ensure: Output matrix $C \in \mathbb{R}^{K \times K}$

```
1: Initialize shared memory tiles for  $A$  and  $B$ 
2: Initialize accumulators for output tile  $C$ 
3: for each  $K$ -tile along the reduction dimension do
4:   Load a tile of  $A$  from global memory into shared memory (asynchronous copy)
5:   Load a tile of  $B$  from global memory into shared memory (synchronous copy)
6:   for each element  $b = B[row, col]$  in the loaded tile do
7:     Scale element by diagonal weight:  $b \leftarrow b \times P[col]$ 
8:   end for
9:   Store the scaled tile of  $B$  into shared memory
10:  Synchronize threads to ensure tiles are ready
11:  Perform warp-level MMA (matrix multiply-accumulate):  $C \leftarrow C + A^T \times B$ 
12: end for
13: Apply epilogue operator (e.g., linear combination with  $\alpha, \beta$ )
14: Write final  $C$  tile to global memory
```

along with the effectiveness of the normalized relative error of Q^{-1} as a practical criterion for selecting the CLP formats. The *Reload* and *Recompute* approaches are compared for M5 and M6 on the RTX3070, M6 on the V100, and M7 on the A100. Finally, we present the results obtained with the SYRK operations, tuned CUTLASS and fused custom GEMM kernel, developed specifically for parametric linear systems.

5.1 Experimental Setup

The experimental evaluation was conducted on a V100 (Volta, compute capability 7.0 with first-generation Tensor Cores supporting FP16), an A100 and a GeForce RTX 3070 (both Ampere, compute capabilities 8.0 and 8.6, respectively with third-generation Tensor Cores that add support for FP64, TF32, BF16, and INT8), and an H100 (Hopper, compute capability 9.0 incorporating fourth-generation Tensor Cores that further introduce FP8 support). The A100 and H100 architectures introduce asynchronous memory copy capabilities. A detailed comparison is provided in Table 4. Experiments were conducted on two systems: (1) a HPC node — equipped with dual AMD EPYC 7702 64-core processors, 384 GB of RAM, and NVIDIA V100, A100, and H100 GPUs; (2) a laptop system — featuring an Intel Core i7-12700H processor, 64 GB of RAM, and a GeForce RTX 3070 Ti GPU. The software environment was built on NVIDIA’s CUDA compiler (nvcc) version 12.4. Dense matrix multiplications were performed using the cuBLAS (v12.4) and cuBLASLt (v12.4) libraries, specifically the cublasDgemm, cublasSgemm, and cublasLtMatmul functions. The CUTLASS library (v3.7) was also employed to develop a customized GEMM kernel tailored for parametric linear systems. For operations specific to these systems, we used a combination of cublasD/Sdgemm functions and custom CUDA kernels. Custom kernels were designed

Table 4 Comparing V100, A100 and H100

Feature	V100 (Volta)	A100 (Ampere)	H100 (Hopper)
Compute Capability	7.0	8.0	9.0
Streaming Multiprocessors (SMs)	80	108	132
CUDA Cores (per SM)	64	64	128
CUDA Cores (Total)	5,120	6,912	16,896
Tensor Cores	640 (FP16)	432 (FP16/BF16/ TF32/INT8)	528 (FP8/FP16/ BF16/TF32/ INT8)
Shared Memory per SM	96KB	164KB	228KB
L1/Shared Cache)	128KB	192KB/128KB	256KB
GPU Memory	32GB HBM2	80GB HBM2e	80GB HBM3
Memory Bandwidth	900GB/s	1.6TB/s	3TB/s

to perform three key tasks: (1) computation of the product PX in FP16 and BF16 precisions, (2) conversion of P matrices into lower-precision formats, and (3) computation of the matrix square root of P ($P^{\frac{1}{2}}$).

Following the recommendations in [24], inner products within Tensor Cores were accumulated in FP32 when using BF16, FP16, and TF32. This enhances numerical accuracy while incurring negligible performance overhead compared to lower-precision accumulation. For OOC computations of G in FP64 or single precision, the cublasXtDgemm and cublasXtSgemm routines from the cuBLASXt library were used. The dynamic range of incoming P matrices was detected online using the cublasIdamax/cublasImin functions. Energy consumption was measured with the NVIDIA Management Library. The linear system $G(P)u = b$ was solved using the Eigen library (<http://eigen.tuxfamily.org>). Finally, the relative error is computed using Equation (4) whereas the *Speedup*, *EnergySaving* and *AveragePowerConsumption* were calculated as the ratios of FP64 precision over SLP for execution time, energy consumption and average power consumption respectively, when solving the parametric linear system.

5.2 Offline Phase

The results of the offline phase of the proposed flow (Algorithm 1), applied to seven matrix sets (Table 1) across four GPUs from three different generations—V100, RTX3070, A100, and H100—are summarized in Table 5. This phase, which can be regarded as the training accuracy, was evaluated using 200 randomly generated P matrices for each precision format (FP32, TF32, BF16, and FP16), equally divided between the P_n and P_w distributions. The first column of the table lists the matrix sets, while the second through fifth columns show the configuration pairs selected for V100, RTX3070, A100, and H100, respectively. A cell containing a single pair indicates that the same configuration was used for all P matrices, whereas a cell with two pairs separated by “/” denotes that the left pair corresponds to P_n and the right pair to P_w .

In summary:

Table 5 Results of applying proposed flowchart for the 7 Matrices - V100, RTX3070, A100, H100. Two results are given if P elements have a narrow (P_n) or wide (P_w) distribution.

Matrices	V100	RTX3070	A100	H100
M1	(FP16,F)	(FP16,F)	(FP16,F)	(FP16,F)
M2	(FP16,T)	(FP16,T)	(FP16,T)	(FP16,T)
M3	(FP32,F)/(FP16,T)	(TF32,F)/(FP16,T)	(TF32,F)/(FP16,T)	(TF32,F)/(FP16,T)
M4	(FP32,F)/(FP16,F)	(FP32,F)/(FP16,F)	(FP32,F)/(FP16,F)	(FP32,F)/(FP16,F)
M5	(FP32,F)/(FP16,F)	(FP32,F)/(FP16,F)	(FP32,F)/(FP16,F)	(FP32,F)/(FP16,F)
M6	(FP32,F)/(FP16,F)	N/A	(FP32,F)/(FP16,F)	(FP64,F)/(FP16,F)
M7	(FP32,F)/(FP16,F)	N/A	(FP32,F)/(FP16,F)	(FP64,F)/(FP16,F)

- For M1, using (FP16,F), all 200 P matrices yield errors below 1% on the V100, RTX3070, A100, and H100.
- For M2, (FP16,T) with normalized P matrices is employed, which also produces errors below 1% across all four GPUs.
- For M3 on the RTX3070, A100, and H100, the approach recommends (TF32,F) for P_n and (FP16,T) with normalized P matrices for P_w . Using (TF32,F) for all P s achieves lower error, while (TF32,F)/(FP16,T) provides better speedup at the cost of higher error and increased GPU memory usage (since X must be stored in both TF32 and FP16 formats). The average errors for the first and second configurations are 0.0016 and 0.0027, respectively. As the V100 does not support TF32, the recommended configurations for M3 is (FP32,F) for P_n and (FP16,T) for P_w .
- For M4 through M7, the proposed configuration for the A100 is (FP32,F) for P_n and (FP16,F) for P_w . For the V100, the suggested configuration is (FP32,F) for P_n and (FP16,F) for P_w . For M4 and M5, the same (SLP,T/F) configurations as those recommended for the A100 are applied to the RTX3070 and H100. However, due to its limited CPU memory capacity, the RTX3070 could not process M6 and M7. And due to higher relative error, for M6 and M7, the suggested configuration for H100 is (FP64,F) for P_n and (FP16, F) for P_w .

5.3 Online Phase

The online phase, which evaluates the application of the selected precision to previously unseen matrices, was tested using a separate set of 10,000 random matrices, equally divided between the P_n and P_w distributions. The results are summarized in Tables 6, 7, 8, and 9, for the V100, RTX3070, A100, and H100, respectively. In each table, the first column lists the matrix sets, while the second column shows the selected configuration pairs. A single pair in this column indicates that the same configuration was applied to all P matrices, whereas two pairs separated by “/” denote that the left pair was used for P_n and the right pair for P_w . The third column reports the average relative error, and the fourth column indicates the number of P matrices that produced an error greater than 1%. The fifth, sixth, seventh and eighth columns present the average speedup, the total number of processed P matrices, the achieved energy and power consumption savings, respectively.

Table 6 Average relative error, speedup, energy and power saving - V100

Matrices	(SLP,T/F)(P_n/P_w)	Average Error	> 1%	Speedup	P/sec	Energy Saving	Power Saving
M1	(FP16,F)	0.00061	2	2.02x	175	2.6x	1.38x
M2	(FP16,T)	0.00038	0	1.95x	70	3.17x	1.68x
M3	(FP32,F)/(FP16,T)	0.00164	0	1.63x	13.5	2.5x	0.86x
M4	(FP32,F)/(FP16,F)	0.0014	0	7.5x	18.1	4.8x	0.6x
M5	(FP32,F)/(FP16,F)	0.0039	5	6.8x	4.3	4.2x	0.6x
M6	(FP32,F)/(FP16,F)	0.0051	7	117.1x	3.7	36.6x	0.33x
M7	(FP32,F)/(FP16,F)	0.0051	0	4.1x	0.05	3.73x	0.6x

On the V100 GPU (Table 6), for M1, two P s exceed the 1% error threshold, with errors of 0.014 and 0.043. For M4, using (FP32,F) for P_n and (FP16,F) for P_w increases the error by 1208%, but improves speedup and energy savings by 33% and 50%, respectively, compared to using (FP32,F) for all P s. For M5, five P s exceed the 1% threshold with an average error of 0.021. Under the mixed configuration of (FP32,F) for P_n and (FP16,F) for P_w , the error rises by 767%, while speedup and energy savings improve by 47% and 50%, respectively, compared to using (FP32,F) for all P s. For M6, seven P s exceed the 1% threshold with an average error of 0.031. The FP64-precision X (32.2GB) exceeds the V100’s memory capacity, forcing out-of-core computation. However, with (FP32,F), the X size is halved to 16.1GB, fitting into memory and achieving an average error of 0.0015, a speedup of 83.2x, and an energy saving of 24.1x. Further applying (FP32,F) for P_n and (FP16,F) for P_w increases the error by 240%, but delivers additional performance gains—41% faster and 52% more energy-efficient—enabling the processing of 3.7 P per second for M6.

For M7, the X sizes in FP64 (80.3GB) and FP32 (40.1GB) both exceed the V100’s memory capacity, making them impractical for real-time processing despite speedup benefits. If all P s are of P_w type, however, FP16 suffices to remain within the error threshold. In this case, the FP16 X (20GB) fits in GPU memory, achieving a 72.3x speedup and enabling the processing of 0.92 P per second. For computing G , *Direct* (see subsections 4.2 and 4.3) is applied for M1–M5, while *Recompute* is used for M6.

Table 7 presents the results for the RTX3070. For M1, two P s exceed the 1% error threshold, with errors of 0.018 and 0.012. Since the RTX3070 Tensor Cores do not support FP64 precision, using FP16 achieves an average 13.4x speedup and 9x energy saving for M1 and M2. For M3 and M4, storing X in FP64 precision exceeds the RTX3070’s memory capacity, requiring out-of-core computation. Switching to FP32 or TF32 allows X to fit in memory, yielding up to 51.5x speedup. For M3, adopting (TF32,F) for P_n and (FP16,F) for P_w causes 42 P s to exceed the error threshold, with an average error of 0.017. Despite a 536% increase in error and 50% higher memory usage compared to (TF32,F) for all P s, this configuration improves speedup and energy saving by 22% and 29%, respectively. For M4, using (FP32,F) for P_n and (FP16,F) for P_w raises the average error by 1665% and memory usage by 50%, but improves speedup and energy saving by 44% and 51%, respectively compared to the configuration that uses (FP32,F) for all P s.

Table 7 Average relative error, speedup, energy and power saving - RTX3070

Matrices	(SLP,T/F)(P_n/P_w)	Average Error	> 1%	Speedup	P/sec	Energy Saving	Power Saving
M1	(FP16,F)	0.00056	2	14.4x	190	10.2x	0.96x
M2	(FP16,T)	0.0011	0	12.35x	74	7.7x	0.8x
M3	(TF32,F)/(FP16,F)	0.007	42	51.6x	11	23.3x	0.46x
M4	(FP32,F)/(FP16,F)	0.003	0	74.3x	11.1	30.3x	0.42x
M5	(FP32,F)/(FP16,F)	0.0012	0	7.2x	0.34	5.8x	0.66x

For M5, storing X in either FP64 or FP32 again exceeds GPU memory, forcing OOC execution and limiting throughput to fewer than 0.4 P /s. By contrast, using FP16 enables X to fit entirely in memory. In the case where all P s are of type P_w , applying (FP16,F) across all P s yields a 167x speedup, processing rate of 8 P /s, and an average error of only 0.23%. For computing G , *Direct* (see subsection 4.1) is applied for M1 and M2, while *Recompute* is used for M3 and M4.

In Table 8 (A100), for M1 only one P exceeds the 1% error threshold, with an error of 0.013. In M3, three P s surpass the threshold, with errors of 0.013, 0.011, and 0.012. For M3, adopting (TF32,F) for P_n and (FP16,T) with normalization for P_w increases the average error by 75% (still below 1%), while improving speedup and energy saving by 10% and 25%, respectively, with a 25% rise in GPU memory usage compared to using (TF32,F) for all P s. Since the P_n distribution is even, half of the P s are handled in (TF32,F) and the other half in (FP16,T).

For M7, four P_n s exceed the 1% error threshold, with an average error of 0.077. Here, storing X in FP64 precision (80.3GB) exceeds the A100's memory, requiring OOC execution. Switching to FP32 reduces X to 40.1GB, enabling a 34.6x speedup and 11x energy saving with an average error of 0.07%. Further adopting (FP32,F) for P_n and (FP16,F) for P_w improves speedup and energy saving by 67% and 72%, respectively, at the cost of a 643% increase in average error (still within acceptable bounds) and 50% higher memory usage compared to (FP32,F) for all P s. If all P s are of type P_w , then (FP16,F) can be applied across all P s, achieving a 201x speedup, throughput of 7 P /s, an average error of 0.44%, and a 67% reduction in memory footprint. For computing G , *Direct* (see subsection 4.1) is applied for M1 to M6, while *Recompute* is used for M7. These results demonstrate that mixed-precision strategies consistently enhance performance on A100, although they introduce moderate increases in error and memory usage.

In Table 9, results on the H100 GPU show that for M1, 14 P matrices exceed the 1% error threshold, with an average error of 0.012. For M2 through M5, the number of P matrices exceeding the threshold is four, three, eleven, and one, with corresponding average errors of 0.020, 0.017, 0.020, and 0.025, respectively. For M6 and M7, using FP32 does not satisfy the 1% error threshold for P_n matrices. Therefore, FP64 has to be used for P_n for M6 and M7 not to exceed the error threshold. Therefore, significant speedup similar to A100 cannot be obtained. For M7, storing X in FP64 precision (80.3GB) exceeds the H100's memory capacity, necessitating OOC execution. Reducing X to FP32 (40.1GB) avoids this limitation but only for P_w matrices. When

Table 8 Average relative error, speedup, energy and power saving - A100

Matrices	(SLP,T/F)(P_n/P_w)	Average Error	> 1%	Speedup	P/sec	Energy Saving	Power Saving
M1	(FP16,F)	0.0004	1	1.76x	439	2.7x	1.6x
M2	(FP16,T)	0.00033	0	1.46x	166	2.4x	1.57x
M3	(TF32,F)/(FP16,T)	0.0028	3	1.46x	41.5	1.9x	1.22x
M4	(FP32,F)/(FP16,F)	0.00084	0	1.63x	31	2.5x	1.5x
M5	(FP32,F)/(FP16,F)	0.00094	0	1.65x	8.5	3x	1.3x
M6	(FP32,F)/(FP16,F)	0.0023	2	1.83x	7.9	2.7x	1.39x
M7	(FP32,F)/(FP16,F)	0.0052	4	57.8x	2	18.9x	0.35x

Table 9 Average relative error, speedup, energy and power saving - H100

Matrices	(SLP,T/F)(P_n/P_w)	Average Error	> 1%	Speedup	P/sec	Energy Saving	Power Saving
M1	(FP16,F)	0.0009	14	1.84x	855	3.8x	2.11x
M2	(FP16,T)	0.0012	4	1.96x	348	3.98x	2.0x
M3	(TF32,F)/(FP16,T)	0.006	3	1.22x	48	2.14x	1.8x
M4	(FP32,F)/(FP16,F)	0.0017	11	2.7x	73.5	4.2x	1.77x
M5	(FP32,F)/(FP16,F)	0.0012	1	3.15x	25.6	4.86x	1.7x
M6	(FP64,F)/(FP16,F)	0.0011	0	1.63x	15.2	1.84x	1.45x
M7	(FP64,F)/(FP16,F)	0.002	0	1.99x	0.065	1.98x	0.46x

all P s are of type P_w , (FP16,F) can be applied across all matrices, achieving a 289x speedup, a throughput of 9.4 P per second, an average error of 0.004, 86.1x energy saving and a 67% reduction in memory usage. For computing G , *Direct* is employed for M1–M6, while *Recompute* is applied for M7.

Table 10 presents the relative error results for the case where the P matrix is an identity diagonal matrix ($P = I$), evaluated on an AMD EPYC 7702 64-core processor, an NVIDIA A100 GPU, and an NVIDIA H100 GPU. As observed, the relative error on the H100 GPU increases with matrix size, indicating reduced numerical stability for larger problems. For comparison, the CPU results were obtained using the Eigen library functions.

For the target system where the relative error of G^{-1} affects the computed solution u accuracy, small values have a greater impact. Consequently, higher arithmetic precision generally results in lower error. Between FP16 and BF16, FP16 provides higher precision due to its 11-bit mantissa, compared to BF16’s 7-bit mantissa. Both formats have sufficient exponent range to handle the values in M1 and M2, so FP16’s superior precision makes it the preferred choice for these cases, as well as for the P_w s in M3–M7.

An examination of data size and value ranges in the X and Q for M1 in Tables 1 and 2, shows that both matrices contain relatively small numbers—less than 0.3 for X and less than 1.6 for Q . Combined with M1’s smaller matrix size, this makes

Table 10 relative error comparison
among CPU, A100 and H100

Matrices	CPU	A100	H100
M1	2.33e-6	6.89e-6	6.46e-6
M2	1.18e-6	1.015e-5	1.09e-5
M3	1.045e-5	5.0e-6	2.78e-5
M4	5.72e-7	2.36e-6	0.000112
M5	5.87e-7	2.39e-6	0.000114
M6	5.87e-07	1.89e-5	0.00088
M7	1.5e-06	1.97e-5	0.00091

Table 11 Relative normalized error of Q^{-1} in BF16 vs. FP16.

Matrices	Q-BF16	Q-FP16	100G-BF16	100G-FP16
M1	0.00265127	0.000176812	0.0032462	0.0002074
M2	0.0376564	0.00137411	0.02098265	0.0010394

FP16 suitable, as it offers sufficient precision and avoids overflow. As the matrix size increases in M2 and the data range becomes wider, normalization of the P s helps retain the suitability of FP16. However, for larger matrices (M3), the increased number of operations, along with a broader data range in X and Q , means FP16 can no longer meet the 1% error threshold. Therefore, TF32 is chosen for H100, A100 and RTX3070, and FP32 is selected for V100. Notably, for P_w s—where values are smaller than in P_n and hence produce smaller entries in G without causing overflow—FP16 remains viable. For the larger matrices M4–M7, which involve even larger Q values and larger X matrices, FP32 is generally selected. Still, for the P_w cases with smaller values, FP16 (and TF32 on H100, A100 and RTX3070) can be used effectively.

5.4 Effectiveness of the normalized relative error of Q^{-1}

To demonstrate that the normalized relative error of Q^{-1} serves as an effective and simple filter for the preliminary selection of candidate low-precision formats—particularly for distinguishing between FP16 and BF16—Table 11 presents both the Q^{-1} relative error and the average G^{-1} relative error across 100 randomly generated P matrices. The results show that BF16 yields a higher Q^{-1} error, which corresponds to a higher G^{-1} average error as well. These findings, consistent across all tested cases, suggest that the normalized relative error of Q^{-1} reliably tracks the average relative error of G^{-1} across different low-precision formats, making it a practical and lightweight filtering criterion.

Table 12 Copy Time vs. Execution Time for cases where X and PX do not simultaneously fit in GPU memory.

Matrices	Data Type	Size of X	GPU	CopyTime (ms)	ExecTime (ms)	Cp/Ex
M3	TF32	4.2	RTX3070	551.5	87	6.3x
M3	FP16	2.1	RTX3070	273.2	55.3	4.9x
M4	FP32	3.8	RTX3070	517.9	116.3	4.5x
M4	FP16	1.9	RTX3070	256.5	41.2	6.2x
M6	FP32	16.1	V100	3242.9	322.6	10.1x
M6	FP16	8.05	V100	1633.8	121.9	13.4x
M7	FP32	40.2	A100	2141.9	806.5	2.7x
M7	FP16	20.1	A100	1073.3	137	7.8x

5.5 Comparing *Reload* vs. *Recompute*

In this section, we compare *Reload* and *Recompute* for those cases where X is larger than GPU memory. Both strategies overwrite X with $XP^{1/2}$. To restore X for each new P , *Reload* reloads X from host memory, whereas *Recompute* recomputes it as $X = X/P^{1/2}$. Table 12 reports the size of X for each benchmark matrix M and selected data type, along with the corresponding copy time from host to device memory, the computation time for evaluating $G = X^\top X$ under different precisions, and the resulting ratio of copy time to execution time ($Cp/Ex = CopyTime/ExecutionTime$). The results clearly indicate that host-to-device copy time dominates execution for large X matrices, making *Recompute* generally more efficient than *Reload*.

Recompute, however, can introduce additional numerical error and degrade the accuracy of X , in particular in lower precision. Table 13 shows the speedup of the two strategies compared to OOC processing, and their average error for various data types. The reported average error is measured over 10,000 P matrices. It is anticipated that repeated multiplication and division of X by $P^{1/2}$ for each processed P matrix will lead to accumulation of rounding errors such that the reconstructed X becomes distinct from the true X in an important way. Over 10,000 multiplications and divisions, *Recompute* does not affect the relative average error when using FP32 or TF32, but it can introduce noticeable error growth in FP16.

Fig. 3 illustrates how the relative average error increases as the number of P matrices grows for the RTX3070, and shows where it exceeds the 1% error threshold. With 100 P matrices, the relative average error is 0.0016 for *Reload* and 0.004 for *Recompute*, becoming 0.0075 for *Recompute* for 600 P matrices with 92 matrices exceeding the 1% error threshold.

To mitigate the FP16 error accumulation, a dynamic reloading strategy can be employed by adapting the number of P matrices processed, namely K , before X is periodically reloaded. After processing K matrices, the accumulated error between the accumulated- X and a freshly reloaded X is evaluated. If it exceeds threshold error (1%) relative to the reloaded X , the value of K is decreased by n ; conversely, if below, K is increased by n . This adjustment balances performance with numerical accuracy by minimizing unnecessary reloads while controlling the growth in error.

Table 13 Performance and Average Error of Reload and Recompute. Speedups relative to out-of-core processing.

Matrices	Data Type	GPU	Speedup Reload	Speedup Recompute	Average Error Reload	Average Error Recompute
M3	TF32	RTX3070	7.6x	42.3x	0.0012	0.0012
M3	TF32/FP16	RTX3070	9.8x	51.6x	0.0062	N/A
M4	FP32	RTX3070	10.2x	51.5x	0.00017	0.00017
M4	FP32/FP16	RTX3070	14x	74.3x	0.001	N/A
M6	FP32	V100	8.5x	83.2x	0.0008	0.0015
M6	FP32/FP16	V100	11.4x	117.1x	0.0051	N/A
M7	FP32	A100	9.8x	34.6x	0.0007	0.0007
M7	TF32	A100	10.8x	51x	0.0018	0.0018
M7	FP16	A100	13.9x	57.8x	0.0029	N/A

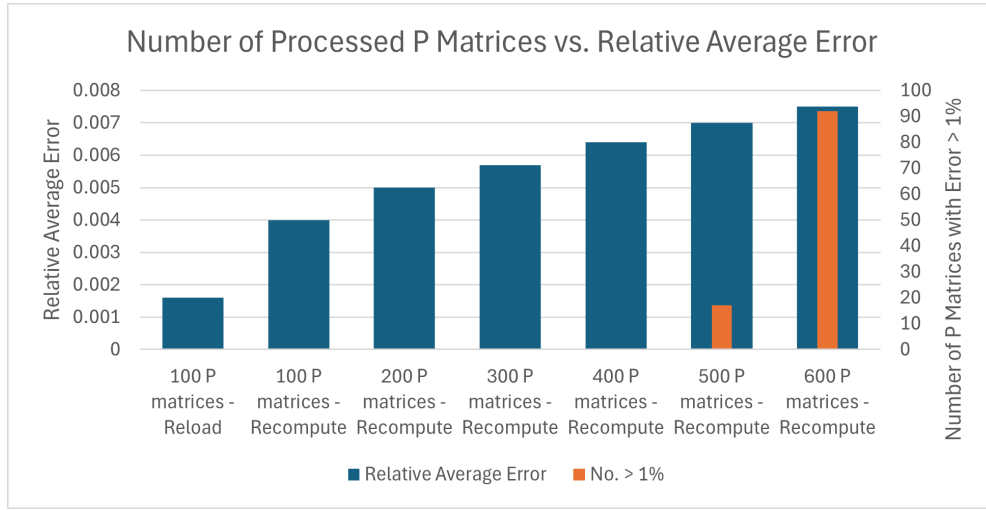


Fig. 3 Effect of number of processed P matrices on the relative average error

5.6 Fused Custom GEMM kernel

The Fused Custom GEMM kernel has the potential for substantial performance improvement when the matrix sizes are such that X and P fit in the GPU memory, but there is not enough space to also store the temporary PX . In this situation, the *Reload* or *Recompute* strategy is required, which incurs substantial overhead. Figure 4 shows the speedup obtained by the Fused Custom GEMM kernel compared to the implementations using two BLAS kernels. For M1 and M2, the two-BLAS implementation uses the Direct strategy. For M3 with lower precision (TF32/FP16), the two-BLAS implementation adopts the Recompute strategy. In contrast, for M3 with double precision,

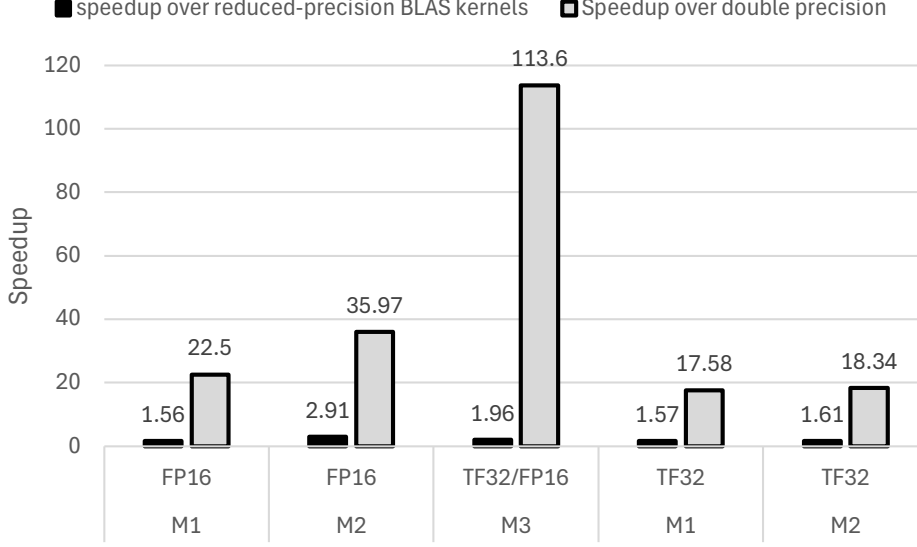


Fig. 4 Speedup achieved by the proposed Fused Custom GEMM Kernel compared to the reduced-precision version and the double-precision baseline, both achieved using two BLAS kernels.

the two-BLAS implementation requires OOC computation due to memory constraints. Results shown are measured on the NVIDIA RTX 3070 GPU.

The Fused Custom GEMM kernel achieves a consistent speedup between 1.5 and 2.9 times compared to a reduced precision implementation with two BLAS kernels. In these results, the configuration of the CUTLASS kernel is tuned to maximise performance on the hardware. The primary tunable parameters include the threadblock, warp, and instruction tile dimensions, pipeline stages, elements per vectorized memory access, and Split-K slices. The average and maximum performance improvement by tuning are 32% and 100%, respectively.

Figure 4 also shows the end-to-end speedup achieved by combining both precision reduction and the custom kernel. Performance is increased by 17-18 $\times$ when using TF32, and up to 113.6 $\times$ when using an adaptive mixture of TF32 and FP16 on the M3 matrix. These results demonstrate that performance optimization on GPUs requires both reasoning about compute speed (which is achieved through transprecision computing) and managing the constraints on GPU memory size (achieved through transprecision computing and custom kernels that eliminate memory transfers).

6 Conclusions

Two orthogonal approaches have been presented to enhance the performance and energy efficiency of parametric linear systems of the form $G(P) = X^T P X$ on modern GPUs. The hybrid coarse-grained transprecision computing methodology accelerates computation by exploiting the numerical error tolerance inherent in many scientific applications. By dynamically selecting lower-precision, GPU arithmetic formats, an

offline phase identifies suitable precision formats and an online phase adaptively selects precision based on the data range of each P matrix. Experimental results show up to 117x speedup and 36.6x energy savings compared to FP64 precision, whilst maintaining an average relative error of only 0.51%. Furthermore, it enables real-time processing of large-scale systems, achieving more than two P -matrix evaluations per second for an 80.3 GB X matrix on an NVIDIA A100 GPU, with relative errors consistently below 1%. The second contribution introduces a fused custom GEMM kernel, which replaces the conventional two-step CUDA-based BLAS computation of $G(P) := X^\top \times P \times X$. The fusion eliminates redundant memory transfers and intermediate computations, providing an additional 3x speedup in G computation. Overall, this work demonstrates the practicality and efficiency of transprecision computing for large-scale, memory-intensive problems.

There are some directions offer promising avenues for extending transprecision techniques toward broader GPU architectures and scientific computing workloads. These include using mixed-precision chunking of X to reduce memory duplication, implementing partial G computation and accumulation across precision levels, developing SYRK kernels optimized for tall-and-skinny matrices at multiple precisions, auto-tuning CUTLASS GEMM configurations for diverse data types and matrix dimensions and investigating GPU-dependent error behavior as matrix sizes scale;

Acknowledgements. The authors acknowledge support from the Engineering and Physical Sciences Research Council under grants, EP/V02860X/1 and EP/V028618/1.

References

- [1] Gilbert, S.: Computational Science and Engineering. Wellesley-Cambridge Press, (2007)
- [2] Lung, R., Wu, Y., Kamilis, D., Polydorides, N.: A sketched finite element method for elliptic models. *Computer Methods in Applied Mechanics and Engineering* **364** (2020)
- [3] Lee, J., Vandierendonck, H., Arif, M., Peterson, G.D., Nikolopoulos, D.S.: Energy-efficient iterative refinement using dynamic precision. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* **8**(4), 722–735 (2018)
- [4] Cherubin, G. S. and Agosta, Lasri, I., Rohou, E., Sentieys, O.: Implications of reduced-precision computations in hpc: Performance, energy and error. IOS Press, 297–306 (2018)
- [5] Kotipalli, P.V., Singh, R., Wood, P., Laguna, I., Bagchi, S.: AMPT-GA: automatic mixed precision floating point tuning for GPU applications. *Proceedings of the ACM International Conference on Supercomputing* (2019)
- [6] Rojek, K.: Machine learning method for energy reduction by utilizing dynamic mixed precision on gpu-based supercomputers. *Concurrency and Computation: Practice and Experience* **31**(6) (2019)

- [7] Ho, N.M., Silva, H.D., Wong, W.F.: GRAM: A framework for dynamically mixing precisions in gpu applications. *ACM Transactions on Architecture and Code Optimization (TACO)* **18**(2), 1–24 (2021)
- [8] Ben Khalifa, D., Martel, M.: Everything you Need to Know About Reduced Mixed Precision Computation in Numerical Programs. working paper or preprint (2023). <https://hal.science/hal-03978176>
- [9] Lin, Z., Sun, A., Zhang, X., Lu, Y.: MixPert: Optimizing mixed-precision floating-point emulation on GPU integer tensor cores. *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (2024)
- [10] Ren, X., Kawai, M., Hoshino, T., Katagiri, T., Nagai, T.: Auto-tuning mixed-precision computation by specifying multiple regions. *Concurrency and Computation: Practice and Experience* **37**(2) (2025)
- [11] Xu, J., Song, G., Zhou, B., Li, F., Hao, J., Zhao, J.: A holistic approach to automatic mixed-precision code generation and tuning for affine programs. *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (2024)
- [12] Gu, R.g., Becchi, M.: GPU-FPtuner: Mixed-precision Auto-tuning for Floating-point Applications on GPU. *IEEE 27th international conference on high performance computing, data, and analytics (hipc)* (2020)
- [13] Cherubin, S., Cattaneo, D., Chiari, M., Di Bello, A., Agosta, G.: TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters* **12**(1), 5–8 (2019)
- [14] Cherubin, S., Cattaneo, D., Chiari, M., Agosta, G.: Dynamic precision autotuning with tafo. *ACM Transactions on Architecture and Code Optimization (TACO)* **17**(2), 1–26 (2020)
- [15] Cattaneo, D., Chiari, M., Fossati, N., Cherubin, S., Agosta, G.: Architecture-aware precision tuning with multiple number representation systems. *2021 58th ACM/IEEE Design Automation Conference (DAC)* (2021)
- [16] Noori, H., Vandierendonck, H., Woods, R., Polydorides, N.: Enhancing the Real-Time Solutions of Parametric Linear Systems on a GPU through Hybrid Coarse-Grained Transprecision Computing. *29th Annual IEEE High Performance Extreme Computing* (2025)
- [17] Chen, J., Xiong, N., Liang, X., Tao, D., Li, S., Ouyang, K., Zhao, K., DeBardeleben, N., Guan, Q., Chen, Z.: TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. *IEEE 27th international conference on high performance computing, data, and analytics (hipc)* (2019)

- [18] Tang, H., Komatsu, K., Sato, M., Kobayashi, H.: Efficient mixed-precision tall-and-skinny matrix-matrix multiplication for GPUs. *International Journal of Networking and Computing* **11**(2), 267–282 (2021)
- [19] Ernst, D., Hager, G., Thies, J., Wellein, G.: Performance engineering for real and complex tall & skinny matrix multiplication kernels on GPUs. *The International Journal of High Performance Computing Applications* **35**(1), 5–19 (2021)
- [20] Rivera, C., Chen, J., Xiong, N., Zhang, J., Song, S.L., Tao, D.: Tsm2x: High-performance tall-and-skinny matrix-matrix multiplication on GPUs. *Journal of Parallel and Distributed Computing* **151**, 70–85 (2021)
- [21] Nakajima, K., Satoh, M., Furumura, T., Okuda, H., Iwashita, T., Sakaguchi, H., Katagiri, T., Matsumoto, M., Ohshima, S., Jitsumoto, H., *et al.*: ppOpen-HPC: open source infrastructure for development and execution of large-scale scientific applications on post-peta-scale supercomputers with automatic tuning (AT). In: *Optimization in the Real World: Toward Solving Real-World Optimization Problems*, pp. 15–35. Springer, (2015)
- [22] Cherubin, S., Agosta, G.: Tools for reduced precision computation: a survey. *ACM Computing Surveys (CSUR)* **53**(2), 1–35 (2020)
- [23] Nvidia Corporation. <https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/> (Sep. 2025)
- [24] Blanchard, P., Higham, N.J., Lopez, F., Mary, T., Pranesh, S.: Mixed precision block fused multiply-add: Error analysis and application to gpu tensor cores. *SIAM Journal on Scientific Computing* **42**(3), 124–141 (2020)