MixPert: Optimizing Mixed-Precision Floating-Point Emulation on GPU Integer Tensor Cores

Zejia Lin
Sun Yat-sen University
Guangzhou, China
linzj39@mail2.sysu.edu.cn

Aoyuan Sun
Sun Yat-sen University
Guangzhou, China
sunaoy@mail2.sysu.edu.cn

Xianwei Zhang*
Sun Yat-sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

Yutong Lu
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn

Abstract

Featuring mixed-precision tensor operations, accelerators significantly enhance performance for many error-tolerant computing tasks, but their applicability is limited in scenarios demanding high precision. While emulating higher-precision data types from lower-precision ones can bridge this gap, existing techniques either struggle to achieve sufficient accuracy or incur excessive overhead, inevitably negating performance gains. To mitigate the issue, we propose `MixPert`, a novel system that balances performance and accuracy via optimizing single-precision emulation on GPU Integer Tensor Cores. `MixPert` devises an efficient data layout and augments the computation pipeline on Tensor Cores. By deeply analyzing performance-precision trade-offs, `MixPert` provides users with multiple configurations based on accuracy requirements. Furthermore, `MixPert` can seamlessly integrate with compilers, facilitating automatic adaptation and tuning of mixed-precision parameters. Evaluations on real-world scientific computing and deep learning applications demonstrate that `MixPert` achieves an average speedup of 1.72× compared to cuBLAS on general-purpose cores. Beyond maintaining improved accuracy, `MixPert` outperforms state-of-the-art approaches APE and CUTLASS by 1.22× and 1.21×, respectively.

CCS Concepts: **• Theory of computation → Massively parallel algorithms**; **• Software and its engineering →** *Compilers.*

Keywords: GPU, Mixed-precision, Emulation, Tensor Core

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LCTES '24, June 24, 2024, Copenhagen, Denmark
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3657567

ACM Reference Format:
Zejia Lin, Aoyuan Sun, Xianwei Zhang, and Yutong Lu. 2024. MixPert: Optimizing Mixed-Precision Floating-Point Emulation on GPU Integer Tensor Cores. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652032.3657567

1 Introduction

Over the past decade, the increasing arithmetic intensity of computational workloads has driven a growing demand for faster processing capabilities. The emergence of error-tolerant applications such as deep learning [39], which prioritize performance over precision, has spurred the development of domain-specific architectures that accelerate general matrix multiply-accumulate (GEMM) in mixed-precision. Accordingly, modern GPUs incorporate the tensor-specialized units featuring such computation, including Nvidia Tensor Cores [8] and AMD Matrix Cores [5]. These tensor architectures are restricted to lower-precision data types like FP16 and BF16, and achieve up to 4× speedup to FP32 computation on general-purpose cores (i.e., CUDA Cores) in high-end GPUs like Nvidia A100 (30× for H200) [10]. To extend this performance benefit to single-precision computation, a plethora of emulation techniques have been proposed. These emulation schemes typically involve splitting and quantizing the original FP32 numbers into segments of types compatible with the hardware constraints like FP16, BF16 or TF32 [17, 33, 36, 37]. A series of low-precision operations are then performed on these segments, and the intermediate results are recombined to produce an FP32 output. However, the inherent rounding and computational errors associated with quantization and low-precision operations accumulate as the matrix size expands, thereby limiting the achievable precision of these emulation techniques compared to native FP32 computations.

Prior attempts to improve mixed-precision GEMM emulation accuracy have either analyzed rounding errors for correction [44] or compromised to the platform-specific format instead of exploring flexible software solutions [7, 47]. These approaches often incur performance overhead and typically lack portability. For instance, on commodity GPUs like the Nvidia RTX3090, half-precision Tensor Cores offer only a 2× speedup compared to FP32 CUDA Cores. Emulation methods requiring a minimum of three half-precision operations would not gain speedup.

In contrast, both modern GPUs comprise substantial performance in INT8 computations within Integer Tensor Cores (ITCs), with speedups ranging from 8× to 60×, making them potentially suitable for emulating higher precision operations across diverse platforms. While integer-based floating-point emulation has been extensively optimized on CPUs through compilers [4] and math libraries [16, 18, 24], these scalar-centric optimizations are not directly applicable to the tensor-oriented architecture of GPUs. Recent work like QuanTensor [33] presents a straightforward approach that quantizes floating-point numbers to INT8 and then refines the results like previous methods. However, this technique requires multiple quantization steps, negating the performance when recovering full accuracy.

To address the limits of existing emulation techniques, we propose `MixPert`, a novel system that leverages high-performance ITCs to accelerate single-precision computations. `MixPert` offers a balance between efficiency and accuracy through a detailed analysis of performance-accuracy trade-offs and multiple configurations for varying error thresholds. The system employs fine-grained data packing and an optimized execution pipeline to enhance performance while maintaining accuracy. Implemented as a compiler pass, `MixPert` seamlessly enables emulation from user-provided source codes. Additionally, it automates mixed-precision parameter tuning based on computational dependencies and user-defined error thresholds, eliminating manual configuration.

In summary, the contributions of this paper are:

- We highlight the inefficiency of balancing performance and precision in existing techniques to emulate high-precision arithmetic on low-precision hardware.
- We systematically analyze the performance-precision trade-offs, and propose `MixPert` to optimize data packing and emulated computing on Tensor Cores.
- We integrate the design with the compiler to facilitate automatic enable code transformation and auto-tuning of mixed-precision configurations.
- The evaluations show that `MixPert` can effectively emulate floating-point arithmetic on Integer Tensor Cores, outperforming the state-of-the-arts while maintaining reduced error levels.

2 Background & Motivation

2.1 Tensor-specialized Hardware

Modern GPUs excel in general-purpose parallel computation thanks to their single-instruction multiple-thread (SIMT) architecture. Domain-specific architectures further push the limits of computation power in accelerating GEMM, an essential operation in many workloads. These tensor-specialized hardware solutions offer remarkable performance advantages, with the trade-off of being limited to pre-defined, lower-precision data types. For example, Nvidia GPUs' Tensor Cores (TCs) do not support the FP32 matrix multiply-accumulate (MMA) of $D = A \times B + C$. Instead, TCs offer mixed-precision compute primitives where A and B must be in FP16, BF16, or TF32 formats, and C, D being FP32 formats. The multiplication occurs in the corresponding input precision, then the intermediate result is upcasted to FP32 and accumulates with the single-precision matrix C, ultimately producing an FP32 output. Similarly, INT8 multiplication is also supported, with intermediate results being accumulated in INT32.

Table 1 details the Nvidia A100 and RTX3090's theoretical peak performance for native data types, including their corresponding exponent and mantissa bitwidth [8, 9]. All listed computations except FP32 utilize Tensor Cores, highlighting the continued absence of FP32 optimization on Tensor Cores in modern GPUs. While low-precision Tensor Cores offer significant performance gains, they come at the cost of potential accuracy degradation and overflow. To enable error-sensitive applications to benefit from this high-performance hardware, software emulation and error control techniques are urgently demanded.

2.2 Extended Data Type Emulation

Emulated data types enable computations on hardware lacking native support by quantizing values into smaller, hardware compatible segments. These segments are processed individually, and the results are recombined to represent the original data type. For instance, a single-precision (FP32) value can be represented as two FP16 values by splitting its mantissa bits as illustrated in Figure 1. The original number is represented by a sign bit s, 8 exponent bits p, and 23 mantissa bits m (including an implied leading one) using a binary radix, as defined by IEEE-754 standard [1] in Equation 1. The

Table 1. Storage format (bits count), underlying computation type, and performance (TFLOPS/TOPS) of different computation types. Upper part: theoretical peak performance. Lower part: measured peak performance.

Data Type	Exp. & Mantissa	Underlying Type	A100 Perf.	RTX3090 Perf.
FP32	8:23	SIMT	19.5	29.8
FP16	5:10	TC	312	59.5
BF16	8:7	TC	312	59.5
TF32	8:10	TC	156	29.8
INT8	-	TC	624	238
cuBLAS [6]	8:23	FP32	17.7	21.7
CUTLASS [7]	8:21	3× TF32	32.0	13.1
APE [36]	8:23	6× BF16	35.6	12.3
QTensor [33]	8:14	3× INT8	N/A	N/A

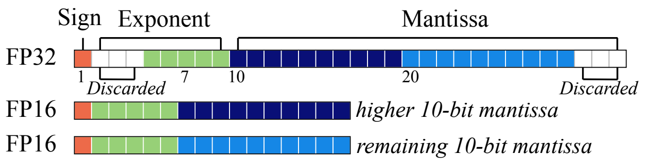


Figure 1. Representing a single-precision value with two half-precision numbers, discarding 6 bits in total.

FP32 value x is split into two FP16 numbers x_{hi} and x_{lo}. The FP32 value is split into x_{hi} and x_{lo}, with x_{hi} containing the higher-order 10 mantissa bits obtained through primitives like `__half`(x). The residual `__half`$(x - x_{hi})$ stores the remaining 10 bits in x_{lo}. This preserves 20 mantissa bits and 5 exponent bits. Multiplication of xy is then computed as $(x_{hi} + x_{lo}) \cdot (y_{hi} + y_{lo})$ using FP16 operations.

$$x = -1^s \cdot (1 + m) \cdot 2^p \tag{1}$$

Building upon this idea of quantization, recent works have explored accelerating single-precision GEMM using low-precision Tensor Cores through emulation. For instance, APE [36] uses six BF16 multiplications to emulate an FP32 multiplication, CUTLASS [7] leverages three TF32 multiplications for emulation, and QuanTensor [33][1] utilizes nine INT8 multiplications with accompanying repetitive scaling and quantization steps. Table 1 summarizes the measured peak performance, and underlying computations involved in the emulation methods, and Figure 2 showcases their speedup on A100 and RTX3090 GPUs, along with the mean relative error E_γ (defined by Equation 10 in §4.1) compared to double-precision (FP64). Both CUTLASS and APE deliver considerable speedup on the A100 platform, achieving 1.98× and 1.90× respectively. This performance boost is attributed to the highly optimized mixed-precision support on A100, which offers 8× to 16× faster performance than FP32. However, this benefit diminishes on the RTX3090, where half-precision acceleration is only 2× and no acceleration for TF32, making these emulation techniques less effective and even negating the performance.

Analyzing accuracy reveals trade-offs between different approaches. While APE [36] faithfully represents an FP32 number using three BF16 numbers, its reliance on six BF16 operations per FP32 multiplication introduces excessive cumulative error inherent in BF16, leading to a high mean relative error of 9.85×10^{-5}. In contrast, CUTLASS [7] achieves lower error than cuBLAS by utilizing the dedicated TF32 format on high-end Nvidia GPUs. Nevertheless, this hardware-specific approach falls short on platforms such as the RTX3090, which lacks optimization for TF32. This underscores its emphasis on leveraging hardware-specific advantages, rather

[1]Denoted as QTensor in the table and figure for brevity. Since QuanTensor is close-sourced and dedicated to Turing architecture, we present the metrics reported in its paper.

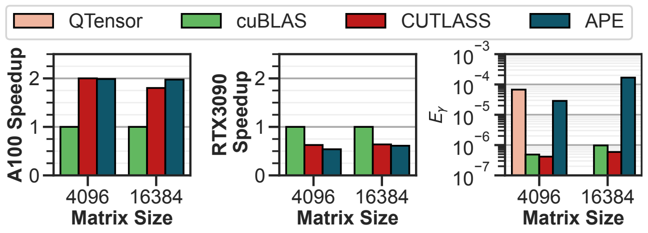


Figure 2. Speedup and mean relative error of GEMM performance on square matrices under different sizes.

than addressing the fundamental issues of accuracy in software emulation. QuanTensor achieves a 2.5× speedup on the RTX2080 Ti, but bears a significant error of 6.76×10^{-5} brought by its scaling quantization from floating-point to 8-bit integer (INT8). Its high-resolution variation, aiming to lower error (1.5×10^{-6}) with progressive residual refinement, serves as a proof-of-concept and comes at the cost of performance penalty (0.83× speedup).

2.3 Challenges to Emulation with Integer

To address the accuracy limitations, one potential solution is to emulate hardware floating-point operations with fixed-point arithmetic directly. This approach has been successfully implemented in CPU-targeted compilers [4] to support embedded systems having no floating-point units, and math libraries [16, 18, 24] to allow for accurate computation in critic domains [40]. This technique separates the exponent and mantissa of a floating-point number into integer representations. For example, multiplying xy can be calculated as $-1^{s_x+s_y} \cdot (1 + m_x) \cdot (1 + m_y) \times 2^{(p_x+p_y)}$, where all multiplications occur in fixed-point arithmetic, eliminating rounding errors. However, porting this fixed-point emulation to ITCs requires addressing unique challenges. ***C1***: ITCs only support INT8 multiplication rather than the widely-used INT32, adding complexities for computing the 23-bit mantissa parts. As ITC only offers 8× speedup on commodity GPUs (60× for high-end), minimizing emulation operations becomes crucial for compatibility. ***C2***: by breaking down FP32 numbers into hardware-compatible segments, efficient data layouts and execution pipelines are essential for maximizing performance. ***C3***: for applications that tolerate mixed-precision computation, the emulation design should offer flexibility for various accuracy configurations to leverage potential performance benefits while meeting various accuracy requirements.

The observations above suggest that prior arts featuring emulation with half-precision floating-point suffer from high accuracy loss and portability issues, while quantizing with fixed-point numbers struggles to balance performance and precision. Distinct from these quantize-and-refine methods, direct emulating with integers holds promise for leveraging the powerful speedup provided by ITC while achieving a superior balance between accuracy and performance for single-precision GEMM computations on GPUs.

3 Design

In this section, we present `MixPert`, a novel approach that addresses the critical challenges of balancing performance and accuracy during the emulation of floating-point operations on ITCs. We first provide a detailed theoretical analysis of this trade-off to guide our approach, then design fine-grained data packing and emulation pipeline optimized for ITCs. Beyond solely designing an emulation algorithm, `MixPert` effectively searches for the optimal configuration for each GEMM operation within an application, considering data dependencies and user-defined precision requirements. This approach enables fine-grained control over the balance of speed and accuracy. Finally, `MixPert` is seamlessly integrated with the compiler, simplifying the process of porting existing applications to leverage the power of this emerging hardware.

3.1 Overview

Figure 3 shows the overall workflow of `MixPert`, which takes source codes as input, and ultimately generates a new executable binary that accelerates floating-point computation through efficient emulation on ITC. The procedure can be divided into four phases around the emulation design and compilation integration: *data packer* to convert the data, *emulation algorithm* to operate on the tensor-specialized hardware, *code transformer* to embed the emulation into source code, and *configuration tuner* to search for optimal settings.

In the emulation design, the *data packer* analyzes and converts input data to fixed-point representations optimized for

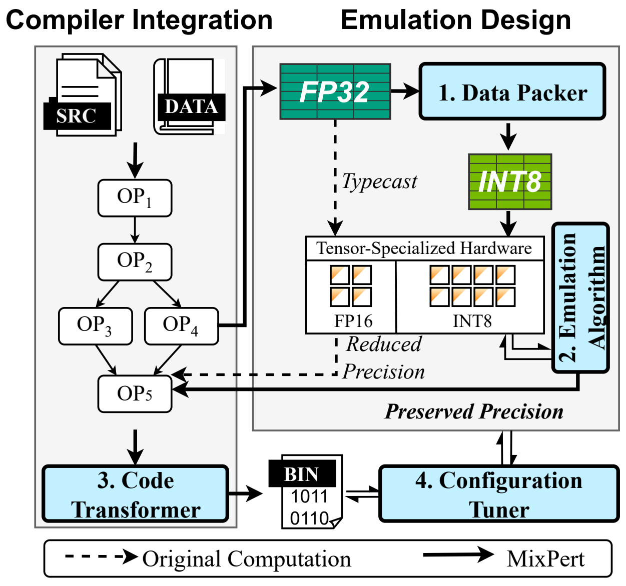


Figure 3. General workflow of `MixPert` to transform reduced precision source code into emulated design with preserved accuracy. An additional tuner works offline to profile and select the proper configuration for each operator.

ITCs, particularly for efficient utilization of the MMA primitives. The *emulation algorithm* then executes emulation of floating-point arithmetic on ITCs using the fixed-point representations, subsequently reconstructing the results back to the original data type. In the compiler integration, *code transformer* is a compiler optimization pass that identifies operations suitable for emulation and replaces them with optimized code segments incorporating data packing and emulation routines. Lastly, the *code transformer* closely works with *configuration tuner* to select the optimal configuration for each emulation operation, aiming to enhance performance under relaxed precision constraints.

The subsequent sections are organized as follows. We first present the representation and emulation of scalar multiply-accumulate operations (§3.2), accompanied by a theoretical analysis of the computational accuracy (§3.2.3). Then we generalize the design to encompass the emulation and optimization of MMA operations on ITCs, including a fine-grained data packing approach (§3.3.2). Finally, §3.4 details the implementation of *code transformer* and *configuration tuner* within the compiler.

3.2 Emulating Scalar Multiply-Accumulate

3.2.1 Scalar Representation. We decompose an FP32 value into four INT8 numbers, preserving all exponent bits and 20 out of 23 mantissa bits. The 8-bit exponent undergoes direct mapping to a single byte. The remaining 20 mantissa bits are divided into three segments, preserving the original value's sign bit. The initial segment captures the crucial higher-order 6 bits, while the subsequent segments handle 7 bits each. Notably, the implied leading one is explicitly stored in the first segment due to its significance in the emulation approach. Figure 4a illustrates the decomposition. Formally, given an FP32 element (a) defined by Equation 1, it is represented as:

$$a \approx -1^s \cdot 2^p \cdot [a_0, a_1, a_2] \cdot [2^{-6}, 2^{-13}, 2^{-20}]^T \tag{2}$$

where a_0, a_1 and a_2 are the segments of mantissa stored in INT8 format. The round-to-nearest mode is adopted for the 21*st* bit, restricting the precision loss to be less than 2^{-21}. While the three least significant mantissa bits are discarded, analysis in Section 3.2.3 demonstrates that this omission is negligible compared to the inherent rounding errors present in standard floating-point computations.

3.2.2 Scalar Multiplication. Consider the multiplication of two FP32 values, x and y, their exponent and mantissa bits are decomposed using Equation 2, yielding two three-dimensional vectors representing their mantissa, denoted as $\boldsymbol{m_x} = [x_0, x_1, x_2]$ and $\boldsymbol{m_y} = [y_0, y_1, y_2]$ respectively. Additionally, their exponent bits are p_x and p_y. The multiplication can be calculated as:

$$x \cdot y \approx -1^{s_x+s_y} \cdot 2^{p_x+p_y} \cdot \boldsymbol{c} \cdot (\boldsymbol{m_x}^T \boldsymbol{m_y}) \cdot \boldsymbol{c}^T \tag{3}$$

where $\boldsymbol{c} = [2^{-6}, 2^{-13}, 2^{-20}]$ for brevity. Notably, this computation involves the outer product of the operands' mantissa vectors $\boldsymbol{m_x}^T \boldsymbol{m_y}$, therefore:

$$x \cdot y \approx -1^{s_x+s_y} \cdot 2^{p_x+p_y} \cdot \boldsymbol{c} \cdot \begin{bmatrix} x_0y_0 & x_0y_1 & x_0y_2 \\ x_1y_0 & x_1y_1 & x_1y_2 \\ x_2y_0 & x_2y_1 & x_2y_2 \end{bmatrix} \cdot \boldsymbol{c}^T$$
$$= -1^{s_x+s_y} \cdot 2^{p_x+p_y} \cdot \sum_{i=0}^{2}\sum_{j=0}^{2} x_iy_j \cdot 2^{-7(i+j)-12} \qquad (4)$$

Each term x_iy_i represents a 13-bit integer[2], contributing to the mantissa bits indexed in $[7(i+j),\ 7(i+j)+12]$.

Figure 4b visualizes the accumulation for terms of x_iy_2, namely $\sum_{i=0}^{2} x_iy_2 \cdot 2^{-7(i+j)-12}$. While *data packer* discards the three least significant bits of the mantissa (represented by x_3 and y_3), these bits are included for illustrative purposes and left-shifted by four to maintain the same bitwidth as the other segments. Each term x_iy_j contributes to specific bit ranges in the final mantissa. For example, x_0y_2 occupies bits indexed in 14-26 (inclusive), and x_1y_2 spans in 21-33. Notably, both x_2y_2 and x_3y_2 fall outside the $28th$ bit, exceeding the achievable precision of 23 mantissa bits in FP32 format. These terms represent inherent rounding errors in native FP32 multiplication, ignoring these terms would not impact the final accuracy within the limitations of FP32 precision. This allows `MixPert` to calculate only eight out of the nine terms in Equation 4. Since INT8 ITC offers 32× speedup compared to FP32 general-purpose cores on A100, this approach yields a theoretical speedup of 4×, surpassing the 2.7× speedup compared to previous works leveraging half-precision-based emulation to achieve the same precision [7, 17, 36].

3.2.3 Accuracy Analysis.

To formally analyze the accuracy of `MixPert`'s emulated multiplication, we define a strict version x' and y' that retains all mantissa bits. This involves incorporating the discarded bits x_3 into the mantissa vector $\boldsymbol{m_x}' = [x_0, x_1, x_2, x_3]$, and the same applies for $\boldsymbol{m_y}'$. Replicating the steps in Equation 4 yields the strict result of $x' \cdot y'$. The absolute error between the emulated and strict versions can be quantified as:

$$e_\delta = 2^{p_x+p_y} \cdot \sum x_iy_j \cdot 2^{-7(i+j)-12}, \quad (i = 3 \text{ or } j = 3) \qquad (5)$$

Leveraging the observation that terms exceeding the $23rd$ bit have negligible impact on final accuracy due to FP32 limitations, we generalize this insight to encompass all terms satisfying $i + j \geq 4$, which correspond to bits beyond the $28th$ position. Consequently, the absolute error term err_δ simplifies to the numerator in Equation 6, and we define the relative error e_γ as:

$$e_\gamma = \frac{(x_0y_3 + x_3y_0) \cdot 2^{-33}}{x' \cdot y'} \qquad (6)$$

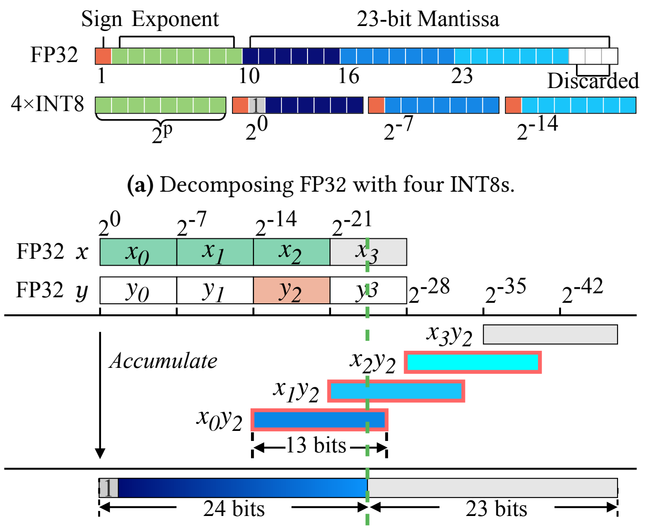
[2]The sign bit is excluded in our analysis for brevity. We assume the multiplication does not produce a carry bit. However, the analysis is also applicable when either the carry bit or the sign bit is present.

We analyze the distribution of e_γ by combining the distribution of individual terms. Each term x_i and y_i is assumed to be independent and uniformly distributed, and the arithmetic operations do not produce a carry bit. The discarded bits are always rounded towards zeros. Specifically, term x_0 uniformly distributes in $[64, \lfloor\sqrt{2^{13}} - 1\rfloor]$, x_1 and x_2 uniformly distribute in $[0, 127]$. Notably, the discarded bits in x_3 are incorporated after left-shifting and applying round-to-nearest in the $20th$ bit, resulting in possible values of 0, 16, 32, or 48. By analyzing the combined distribution of all terms, the mathematical expectation of e_γ is approximately 2.82×10^{-7}, well below 1.5×2^{-22}. In worst case, $e_\gamma \approx 7.15 \times 10^{-7} < 3 \times 2^{-22}$.

3.2.4 Scalar Accumulation and Accuracy.

Adding up floating-point numbers with identical exponents involves directly summating their mantissa parts. However, when the exponents differ, an alignment process should precede the summation. This crucial step involves right-shifting the mantissa of the smaller number by the absolute difference between the exponents, ensuring the preservation of the original value's fidelity. Specifically, the implied leading one demands consideration during alignment and summation and *data packer* already explicitly stored it. Following the mantissa addition, the resulting value requires shifting and normalization to comply with the floating-point format.

Figure 5 illustrates the concept of this exponent alignment. Consider three elements $x^{(0)}$, $x^{(1)}$ and $x^{(2)}$, with respective exponents p, $p+4$ and $p+2$ and mantissas m_1, m_2 and m_3. Since the maximum exponent is $p+4$, to align the exponent of v_1 to $p+4$, its mantissa bits need to be shifted by four



(a) Decomposing FP32 with four INT8s.

(b) Part of the emulated multiplication in Equation 4. The accumulated terms $\sum_{i=0}^{2} x_iy_2 \cdot 2^{-7(i+j)-12}$ are highlighted in red edges. Gray blocks suggest discarded values.

Figure 4. Representing scalar and emulating its multiplication with INT8 operations.

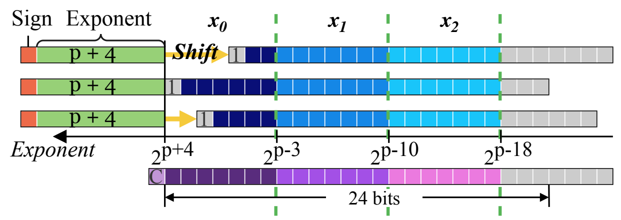


Figure 5. Uniforming exponent bits and shifting mantissa for accumulation when the exponent bits are different.

positions, resulting in the representation $(1 + m_1) \cdot 2^{-4} \cdot 2^{p+4}$. Similarly, v_3 is right-shifted by two positions and represented as $(1 + m_3) \cdot 2^{-2} \cdot 2^{p+4}$. The *data packer* splits the mantissa into three fragments, which are denoted as $\boldsymbol{x}_0$, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. In the example, the summation produces a carry bit, to convert the result back to valid FP32 format, the carry bit is encoded as the implied leading one and the output exponent is $p + 5$. The mantissa bits that exceed the 24*th* bit are rounded to the 23*rd* bit. Since this approach strictly emulates the FP32 scalar accumulation, no extra error is introduced compared to native floating-point computation.

3.3 Emulating Matrix Multiply-Accumulate

Leveraging ITC requires decomposing the matrix into tiles of hardware-compatible sizes (e.g. 16×32 and 32×8), and invoking the MMA primitives to perform an atomic operation within each tile. Therefore, adapting the emulation to these units necessitates pre-processing beforehand, including packing data to INT8 format and shifting mantissa to align exponent, adding complexity to the software pipeline. In this section, we begin by extending the emulated scalar-scalar computation to encompass vector-vector operations between n-dimensional vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Matrix multiplication can be easily derived from this concept, as each element in the result matrix is essentially the inner product of its corresponding row and column vectors. We then demonstrate how the emulated computation operates within each tile and convert the results back to valid FP32 formats. Finally, we analyze the accuracy of the emulated matrix multiplication.

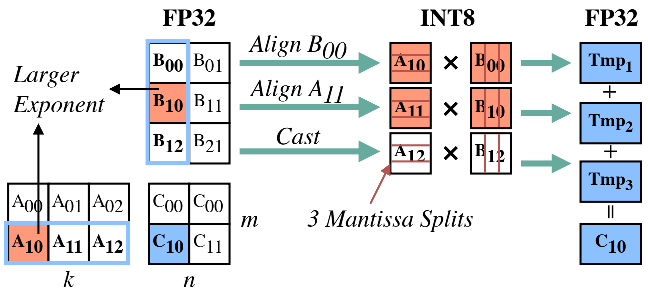


Figure 6. Example to compute C_{10} by decomposing matrix and aligning exponent within different tiles. Matrix A's mantissa segments are stored as row-major, and column-major for B's mantissa segments.

3.3.1 Vector Multiplication and Tuning Knobs.

Representing a vector efficiently requires aligning the exponent bits of its elements for subsequent accumulation operations. Similar to the concept of exponent alignment of individual values in Figure 5, an n-dimensional vector $\boldsymbol{x}$ can be expressed as:

$$\boldsymbol{x} \approx 2^{p_{\max}} \cdot [\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2] \cdot [2^{-6}, 2^{-13}, 2^{-20}]^T \tag{7}$$

where the sign bits are omitted for brevity, $p_{\max}$ is the maximum exponent value in the vector's elements, $\boldsymbol{x}_i$ is the n-dimensional INT8 vector containing the mantissa split of the elements, each shifted by the difference between its corresponding element's exponent and $p_{\max}$. To emulate n-dimensional vector-vector multiplication $\boldsymbol{x} \cdot \boldsymbol{y}$, $p_{\max}$ is further set to the maximum exponent value in the two vectors, and the multiplication can be derived as:

$$\boldsymbol{x} \cdot \boldsymbol{y} \approx 2^{p_{\max}} \cdot \sum_{i=0}^{l} \sum_{j=0}^{r} \boldsymbol{x}_i \boldsymbol{y}_j^T \cdot 2^{-7(i+j)-12} \tag{8}$$

where l and r respectively denotes the number of splits minus one by *data packer*. Equation 7 splits $\boldsymbol{x}$ into three segments, so l and r are set to two. When $l = r = 3$, no mantissa bits are discarded, resulting in full precision emulation, as mentioned in §3.2.3. Strategically adjusting l and r based on the error tolerance allows performance improvements to be traded with reduced accuracy. For instance, setting $l = 2$ and $r = 1$ implies discarding the least significant 3 mantissa bits from $\boldsymbol{x}$ and the least significant 10 bits from $\boldsymbol{y}$. This configuration reduces the required INT8 multiplications to six, achieving a speedup of 1.3× compared to the default emulation configuration. We define four levels of configuration I - IV, denoting the combinations of (l, r) as $(1, 1)$, $(1, 2)$, $(2, 1)$ and $(2, 2)$.

3.3.2 Tiled Data Packing and Computing.

For efficient pipeline to emulate matrix multiply-accumulate of $C = A \times B$, `MixPert` begins by partitioning the input matrices into hardware-compatible tiles denoted as A_{ik}, B_{kj}, and $C_{ij} = \sum_k A_{ik} B_{kj}$. For each tile pair, the algorithm identifies the highest exponent present in both operands involved in the multiplication within that specific tile. Each mantissa value within the tile is shifted based on the maximum exponent of its corresponding operand. The mantissa values are efficiently packed into the INT8 format following the exponent-based shifting. This pre-processed data is then fed into the hardware's MMA primitive to perform the actual computation within each tile and produce an INT32 number. Finally, the intermediate results are converted back to valid FP32 format by bit operations to incorporate exponent bits and shift mantissa bits. These results are accumulated to yield the final output matrix C.

As an illustrative example in Figure 6, where matrix A and B are decomposed into 2×3 tiles and 3×2 tiles, respectively, resulting in matrix C composed of 2×2 tiles. In this specific case, tiles A_{10} and B_{10} have larger exponents than others. To emulate multiplication, we need to shift B_{00} to

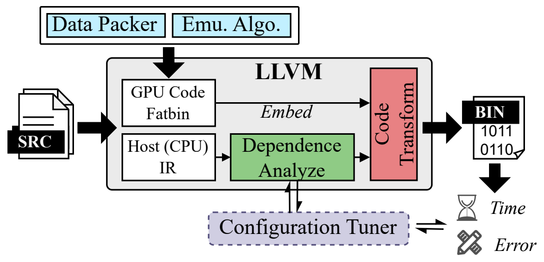


Figure 7. Compiling and precision tuning with MixPert.

align its exponent with A_{10} and similarly adjust A_{11} with B_{10}. After performing the MMA instruction, the resulting INT32 value is converted to FP32 intermediate value Tmp_i and is accumulated to yield the final result C_{10}. Notably, for the intermediate result $A_{00}B_{00}$ corresponding to C_{00}, both operands share the same exponent, obviating the need for further shifting. This tile-based shifting strategy allows for a fine-grained approach to data conversion, thereby limiting the potential for error within individual tiles.

3.3.3 Accuracy Analysis. In standard floating-point arithmetic, the inner product of two k-dimensional vectors incurs an error bound of ku, where u represents the unit round-off (e.g., 2^{-24} for FP32 numbers) [12]. This bound arises due to rounding errors accumulated during the summation process, meaning that the least significant $\delta = \lfloor \log_2 k \rfloor$ bits of the mantissa are rounded. During exponent alignment in the emulation algorithm, the mantissa smaller operand is right shifted by δ', the difference between its exponent and the maximum one, discarding the least significant δ' bits. However, if $\delta' \leq \delta$, this discarding does not introduce additional error as the term falls within the inherent error bound of floating-point representation. For any element x_i with $\delta'_i > \delta$, its error contribution is $(\delta' - \delta)u/k$. The total error accumulates as:

$$\Delta = \sum_{i=0}^{k} \frac{(\delta'_i - \delta)u}{k} \tag{9}$$

In practice, the MMA instruction performs $16 \times 8 \times 32$ matrix multiplication, i.e., 16×32 the left operand and 32×8 the right operand. With $k = 32$ as the accumulated dimension, the emulation achieves higher accuracy than native FP32 computation if the largest element in the tile is not more than 32 times the smallest. The locality of data distribution fulfills this requirement in most cases. However, to handle outliers or excessively large distribution ranges, the *data packer* identifies these problematic tiles during pre-processing and switches to a more robust half-precision-based emulation.

3.4 Compiler Integration and Tuning

Applications manifest varying levels of computational complexity and domain-specific accuracy requirements. To alleviate the programming burden when applying MixPert to diverse existing applications, we compose a compiler pass for code transformation and a corresponding tuner to search for the optimal configuration. This integration automatically embeds the emulation design into the source code during compilation, easing the effort of optimizing these mixed-precision applications.

Figure 7 shows the implementation framework of MixPert, based on LLVM Compiler Infrastructure [32]. The source code for GPU kernels is separated from the host (CPU) code and individually compiled by the hardware vendor's compiler (e.g., Nvidia's nvcc) into binary code. Meanwhile, the compiler's front-end compiles the host code into an intermediate representation (IR). This step also applies to the source code of the *data packer* and *emulation algorithm* implementations. Following this separation, the *code transformer* compiler pass analyzes data dependencies amongst the GEMM operations and assigns them unique identifiers. The dependencies are then fed to the *configuration tuner* to facilitate the analysis of error propagation among operators. Furthermore, the transformation pass substitutes the original operators with our emulated versions by embedding the corresponding GPU binary code into the IR. This concludes the role of *code transformer* and the remaining compilation steps are handled by the underlying compiler.

To address the need for reduced-precision computation in applications tolerant of relaxed accuracy constraints, we introduce a lightweight tuner that optimizes the tuning parameters in Equation 8 (§3.3.1). Initially, the system profiles the program, gathering both execution time and mean relative error (Equation 10). Then it utilizes a greedy algorithm to incrementally decrease the precision of operators, guided by the execution time predominance of each operator. If precision reduction compromises output quality, the tuner attempts to elevate the precision of the immediate predecessor if its execution time is shorter. This strategy maintains acceptable output quality while allowing for potential precision reduction in time-consuming operators.

4 Experimental Evaluation

4.1 Methodology

Platforms: We conduct experiments on a server equipped with Nvidia A100-PCIe-40GB GPUs, an AMD EPYC 7742 64-Core CPU and 256GB DRAM. The operating system is Debian 5.10.179 and the version of the Nvidia driver is 545.23.08. We use LLVM 15.0.7 to implement our code transformation pass, and the emulation algorithm is materialized with CUTLASS 3.2.1. All GPU programs are compiled using CUDA 11.8.0 with the compiler option -O3 is switched on.

Evaluated Schemes: We compare MixPert against several prior arts, including APE (emulation with BF16) [36] and CUTLASS (emulation with TF32) [7]. Both APE and CUTLASS are configured in accordance with their officially suggested parameters for the Ampere architecture [22, 35]. We take cuBLAS's FP32 implementation [6] as the baseline and normalize the performance to it.

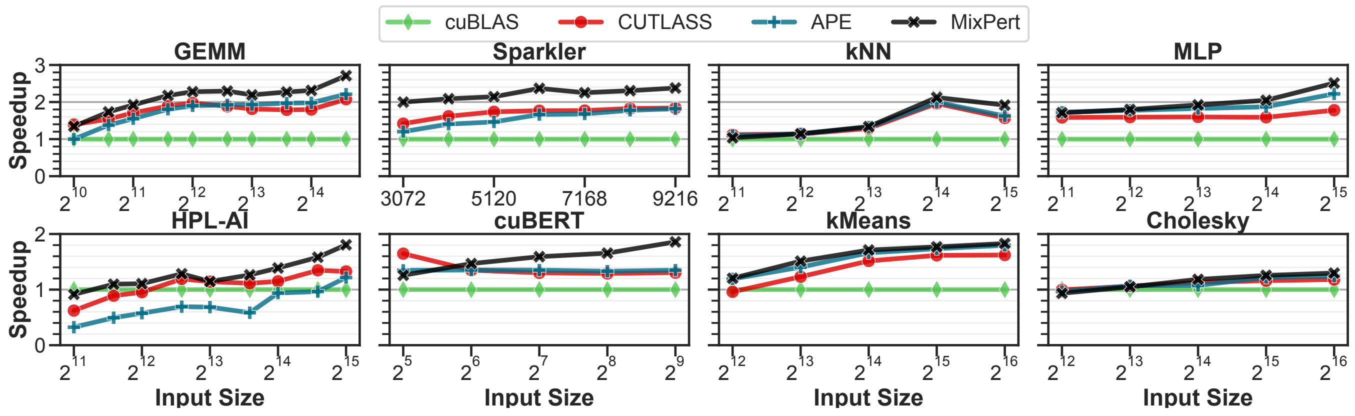


Figure 8. End-to-end performance of different floating-point computation schemes.

Benchmarks: We conduct experiments using six real-world applications and a custom-built micro benchmark. Their domains include linear algebra, genomics computing, machine learning and deep learning, as briefly described below. HPL-AI [30] is a popular benchmark to assess the mixed-precision performance of accelerators, leveraging the High-Performance Linpack (HPL) [13] algorithm. Cholesky factorization [26] is a fundamental linear solver used to decompose a matrix. Sparkler [27] is a mini-application employed in the CoMet comparative genomics application [28] for calculating the correlation coefficient of genes. K-Nearset Neighbors (kNN) [19] and K-means clustering [29] are widely used statistical machine learning methods for classification and clustering. cuBERT [15] is a large language model based on the Bidirectional Encoder Representations from Transformers (BERT) [14]. Our micro benchmark contains a custom-built three-layer multilayer perceptron (MLP) [41], and a single GEMM operation.

Metrics: For performance measurement, we average the execution time of all GPU kernels with 15 repeated runs. The timings include all overhead introduced by the emulation scheme and exclude data transfer time between GPU and CPU. To evaluate the accuracy, we use mean relative error E_γ defined with Frobenius norm, where X is the ground-truth matrix and X^* is the emulated result:

$$E_\gamma(X, X^*) = \frac{\|X - X^*\|_F}{\|X\|_F} = \frac{\sqrt{\sum (X_i - X_i^*)^2}}{\sqrt{\sum X_i^2}} \tag{10}$$

4.2 Performance

We conduct the evaluations with input consumption of around 1GB to 15GB of memory for each application. The benchmarks' input parameters are configured to be optimal with cuBLAS. Figure 8 shows the end-to-end performance results. Overall, `MixPert` effectively accelerates applications, outperforming APE and CUTLASS in most cases. Averaging across benchmarks and input sizes, `MixPert` achieves a 1.72× speedup compared to cuBLAS, while APE and CUTLASS achieve 1.41× and 1.45×, respectively. This confirms the effectiveness of using Integer Tensor Cores for floating-point computations. Notably, `MixPert` excels in square matrix GEMM across all input sizes, with performance scaling well and reaching a 2.71× speedup at an input size of 24576. This is because the overhead of data packing and restoring becomes less significant compared to the computation time for larger matrices. Sparkler, with its larger k dimension than m and n, further benefits from reduced overhead, consistently achieving at least a two-fold speedup. While CUTLASS and APE struggle with HPL-AI's tall and thin matrices (i.e., $k \ll m$ and n), `MixPert` leverages HPL-AI's mixed-precision computation by adopting lower precision for calculations. Although this cannot fully cover the overhead in the smallest input size, `MixPert`'s speedup steadily increases as the matrix size grows. Overall, `MixPert` demonstrates broad applicability and performance benefits across diverse domains.

4.3 Accuracy

In this section, we study the accuracy of the emulation methods. Figure 9 shows the Frobenius error of four representative applications, each launched with the same parameters in Figure 8. We compare the error of GEMM and HPL-AI with

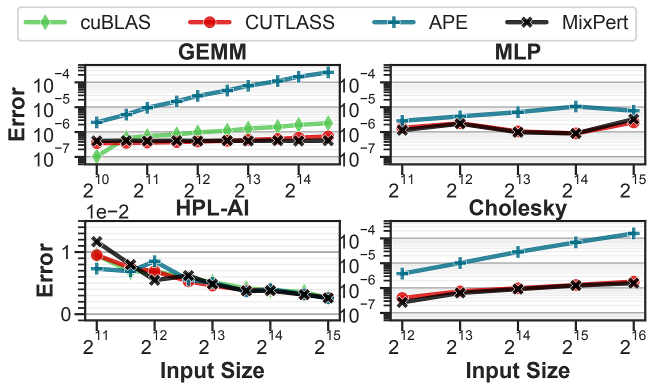


Figure 9. Output error of four benchmarks with the identical launch configurations in Figure 8.

double-precision (FP64) computation to provide a comparison with cuBLAS's standard FP32 computation, and report the error of MLP and Cholesky compared to FP32. In average of all input sizes, the error of MixPert is significantly lower, at 4.46×10^{-7}, compared to cuBLAS (1.14×10^{-6}), CUTLASS (4.61×10^{-7}) and APE (7.16×10^{-5}). One key factor contributing to the higher error in cuBLAS and APE is the accumulation of errors during intermediate result calculations in low-precision, especially for larger matrices. MixPert effectively mitigates this issue through its fixed-point emulation approach, which reduces the error introduced by floating-point operations.

HPL-AI employs the GMRES [45] precision refinement technique to recover the final solution to 64-bit accuracy. The accuracy is measured using a scaled residual, which needs to be less than 16 for valid output. Due to this refinement step, all methods achieve similar accuracy levels. However, MixPert leverages this opportunity for mixed-precision computation by reducing the number of mantissa splits in its emulation, thereby improving performance without compromising accuracy. Similar to GEMM, the errors in MLP and Cholesky factorization for MixPert are well-maintained around 10^{-6} and are robust against larger input sizes. This demonstrates MixPert's capability to achieve high accuracy while offering performance benefits.

4.4 Tuning Efficiency

Figure 10 showcases the performance improvements achieved by the lightweight tuner when we relax error constraints. Performance is measured as speedup normalized to cuBLAS, while accuracy is compared against cuBLAS's results. We evaluate the tuner in two scenarios. The Cholesky factorization with an input size of 32768 and four GEMM dependencies, and MLP with an input size of 16384 and three dependent GEMM operations. In both cases, MixPert maintains the error within the specified threshold while achieving speedups comparable to those obtained through an exhaustive search. When the threshold is relaxed to 5×10^{-4} for Cholesky factorization, all the precision levels are lowered to the lowest one and gain 25% performance improvement compared to the original precision of MixPert. This exemplifies the effectiveness of the tuner in identifying optimal precision configurations that balance performance and accuracy.

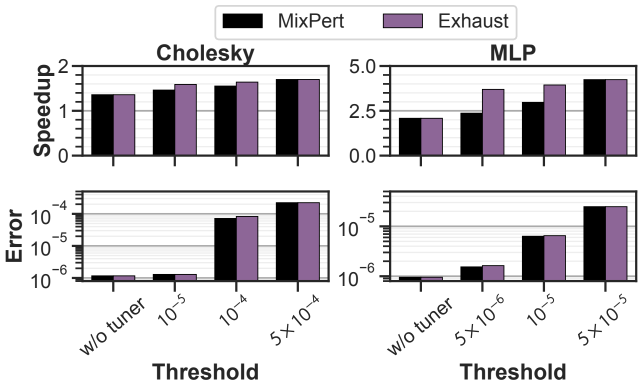


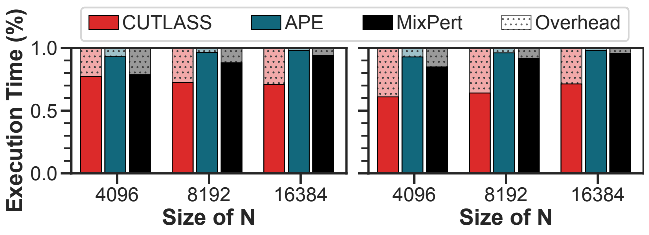
Figure 10. Speedup and error of Cholesky factorization and MLP being optimized by the tuner.

4.5 Performance Breakdown

To examine the overhead brought by the emulation algorithms, we profile the programs and analyze their breakdown in Figure 11. MixPert has an average overhead of 11.1%, APE and CUTLASS is 4.29% and 30.5% respectively. The primary contributor to this overhead in MixPert is the conversion of intermediate INT32 results back to FP32 during accumulation. However, as the matrix size increases, this overhead becomes less significant. For example, MixPert's overhead reduces to 4.24%, manifesting minimal impact on performance for larger matrices.

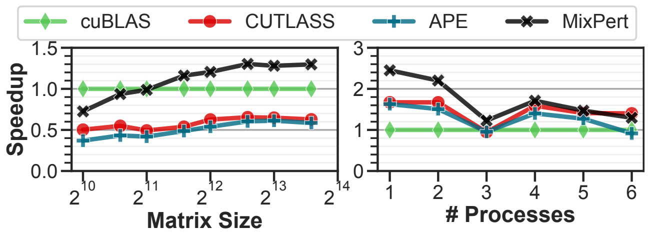
4.6 Sensitivity Study

We evaluate how Tensor Core's performance impacts the emulation. Figure 12a shows square matrix GEMM's performance on an Nvidia RTX3090 GPU, which features less powerful half-precision Tensor Cores compared to the A100 as detailed in Table 1 (§2). Due to the lack of support for half-precision, both CUTLASS and APE are unable to outperform cuBLAS on this platform. In contrast, MixPert maintains a 1.12× average speedup over cuBLAS, which even increases to 1.30× with larger matrices. This demonstrates MixPert's adaptability to different hardware capabilities, suggesting that MixPert can also offer performance benefits on other commodity accelerators that prioritize INT8 performance.



(a) Matrix Size ($N \times N \times N$). **(b)** Matrix Size ($N \times N \times 2N$).

Figure 11. Execution time breakdown of GEMM.



(a) Performance of $N \times N \times N$ GEMM on Nvidia RTX3090. **(b)** Performance of Sparkler under various co-locating processes.

Figure 12. Sensitivity study on different platforms and resource constraints.

To evaluate the performance robustness of MixPert under constraint system resource, specifically when multiple processes share a single GPU, we measure Sparkler's performance with varying Message Passing Interface (MPI) processes on one GPU. The results presented in Figure 12b demonstrate that speedup decreases with increasing processes due to limited Tensor Core availability. However, MixPert maintains significant advantages, outperforming CUTLASS by 24.3% and APE by 33.9% with five or fewer processes, showcasing its efficient integer Tensor Core utilization even in shared environments.

4.7 Discussion

The evaluations demonstrate MixPert's efficacy on existing Nvidia GPUs , and MixPert can also extends to future generations [10] or other hardware vendors [5, 46]. These platforms offer enhanced integer tensor capabilities, boasting up to a 2× speedup compared to their half-precision counterparts. MixPert leverages its unique focus on integer operations to achieve a different balance between performance and accuracy, which is not well explored by previous emulations using half-precision [7, 17, 36]. Moreover, MixPert avoids the error and overhead associated with triple times of quantization to represent FP32 with INT8s [33], opting instead for a more efficient splitting and recombination strategy.

While MixPert's integer-focused emulation excels in many scenarios, it acknowledges several limitations. First, representing data characterized by highly deviated distributions introduces substantial absorption errors for relatively small values (§3.3.3). MixPert falls back to the conventional half-precision-based emulation methodology to maintain accuracy. This adaptive strategy incurs a marginal performance overhead in comparison to techniques that solely implement half-precision emulation. Second, the emulation overhead becomes non-negligible when performing GEMM operations on small matrices. This challenge remains an open question in prior arts [33, 36], and MixPert does not yet offer a definitive solution. Finally, MixPert uses mean squared relative error as the error metric, further investigation is needed to customize it for application-specific metrics. Nonetheless, MixPert presents a compelling alternative approach, demonstrating tangible performance improvements, particularly for large-scale data exhibiting a concentrated distribution.

5 Related Works

5.1 Mixed-precision Emulation

Mixed-precision emulation has been extensively explored to enable arithmetic operations that lack native hardware support. On GPUs, Markidis [37] has pioneered using TF32 operations for GEMM emulation on Tensor Cores, with EGEMM-TC [17] and CUTLASS [7] further optimizing for this format. QuanTensor [33] and APE [36] broaden support for data types and emulation strategies. Ootomo [42] analyzes and corrects rounding errors in quantum computing, using up to 91 operations to emulate double-precision GEMM [43]. Frameworks like TVM [11] and QGTC [48] extend emulation to arbitrary-bitwidth integers on Tensor Cores, specifically targeting deep learning workloads. On CPUs, the GCC Compiler [4] has long supported emulation for embedded systems lacking floating-point units. Math libraries like MPFR [18], SoftFloat [24], and CPFloat [16] further optimize emulation performance on CPUs. Distinguishing from previous works, MixPert directly emulates floating-point computation on Tensor Cores without quantization, well-preserving accuracy while achieving high performance.

5.2 Mixed-precision Tuning

Mixed-precision tuning has gained popularity as applications tolerate relaxed error thresholds [2, 23, 34]. It has been extensively explored targeting both CPU [3, 21] and GPU [20, 31] applications, well accounted for code pattern and precision constraints. ADAPT [38] estimates errors based on operator dependencies, while Gram [25] dynamically selects precision with high overhead. FPLearner [49] predicts error and performance using a deep learning model, and PrecTuner [50] optimizes mixed-precision computation within nested loops. MixPert differs from these dedicated tuners by focusing on its inherent tunable parameters, offering a lightweight approach that can be combined with existing tuners for joint optimization with the emulation.

6 Conclusion

The paper proposes MixPert to accelerate floating-point emulation on GPU Integer Tensor Cores while keeping the accuracy of FP32. MixPert efficiently packs data at a granular level and optimizes the processing pipeline specifically for Tensor Cores. Additionally, MixPert conducts in-depth analysis of the trade-offs between performance and precision, providing more configurations to exploit potential performance gains under relaxed error constraints. Finally, MixPert is integrated with compilers to facilitate automatic emulated design and fine tuning of mixed-precision parameters. Experimental results demonstrate that MixPert effectively balances accuracy and performance, outperforming the state-of-the-art methods by 1.21×.

Acknowledgments

We express our sincere gratitude to the anonymous reviewers for their constructive feedback. This research was supported by the National Natural Science Foundation of China-#62332021, Major Program of Guangdong Basic and Applied Research-#2019B030302002, Key-Area Research and Development Program of Guangdong Province (No.2021B0101190003), CCF-Phytium Fund (CCFPhytium202204), and the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (23xkjc016).

References

[1] 2019. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] Ahmad Abdelfattah, Hartwig Anzt, Erik G. Boman, Erin C. Carson, Terry Cojean, Jack J. Dongarra, Alyson Fox, Mark Gates, Nicholas J. Higham, Xiaoye S. Li, Jennifer A. Loe, Piotr Luszczek, Srikara Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry F. Smith, Kasia Swirydowicz, Stephen J. Thomas, Stanimire Tomov, Yaohung M. Tsai, and Ulrike Meier Yang. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *Int. J. High Perform. Comput. Appl.* 35, 4 (2021). https://doi.org/10.1177/10943420211003313

[3] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. *ACM SIGPLAN Notices* 52, 1 (2017), 300–315. https://doi.org/10.1145/3009837.3009846

[4] GNU Compiler Collections. 2022. Soft float library routines. https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html.

[5] AMD Corporation. 2023. Amd cnda 3 architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[6] Nvidia Corporation. 2014. Cuda basic linear algebra subroutine library (cublas). https://docs.nvidia.com/cuda/cublas/index.html.

[7] Nvidia Corporation. 2018. Cuda templates for linear algebra subroutines (cutlass). https://github.com/NVIDIA/cutlass.

[8] Nvidia Corporation. 2021. Nvidia A100 tensor core GPU architecture. https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper.

[9] Nvidia Corporation. 2021. Nvidia ampere ga102 gpu architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf.

[10] Nvidia Corporation. 2022. Nvidia H100 tensor core GPU architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[11] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, 305–316. https://doi.org/10.1145/3368826.3377912

[12] Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Theo Mary, and Mantas Mikaitis. 2022. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science* 9, 3 (2022), 211631. https://doi.org/10.1098/rsos.211631

[13] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. 2011. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the 25th International Conference on Supercomputing*. ACM, 162–171. https://doi.org/10.1145/1995896.1995923

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American*. Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[15] Liwen Fan, Ruixin Wang, Kuan Fang, and Xian Sun. 2019. Cubert. https://github.com/zhihu/cuBERT.

[16] Massimiliano Fasi and Mantas Mikaitis. 2023. Cpfloat: a c library for simulating low-precision arithmetic. *ACM Trans. Math. Softw.* 49, 2 (2023), 18:1–18:32. https://doi.org/10.1145/3585515

[17] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming*. ACM, 278–291. https://doi.org/10.1145/3437801.3441599

[18] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. Mpfr: a multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (2007), 13–es. https://doi.org/10.1145/1236463.1236468

[19] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. 2010. K-nearest neighbor search: fast gpu-based implementations and application to high-dimensional feature matching. In *Proceedings of the International Conference on Image Processing*. IEEE, 3757–3760. https://doi.org/10.1109/ICIP.2010.5654017

[20] Ruidong Gu and Michela Becchi. 2020. Gpu-fptuner: mixed-precision auto-tuning for floating-point applications on gpu. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics*. IEEE, 294–304. https://doi.org/10.1109/HIPC50609.2020.00043

[21] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 333–343. https://doi.org/10.1145/3213846.3213862

[22] Manish Gupta. 2022. FP32 emulation via tensor core instruction. https://github.com/NVIDIA/cutlass/tree/main/examples/27_ampere_3xtf32_fast_accurate_tensorop_gemm.

[23] Azzam Haidar, Stanimire Tomov, Jack J. Dongarra, and Nicholas J. Higham. 2018. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE, Article 47, 11 pages. https://doi.org/10.1109/SC.2018.00050

[24] John Hauser. 2016 (Retrieved 2024.1). Berkeley softfloat. http://www.jhauser.us/arithmetic/SoftFloat.html.

[25] Nhut-Minh Ho, Himeshi De silva, and Weng-Fai Wong. 2021. Gram: a framework for dynamically mixing precisions in gpu applications. *ACM Trans. Archit. Code Optim.* 18, 2 (2021), 1–24. https://doi.org/10.1145/3441830

[26] Emmanuel Jeannot. 2012. Performance analysis and optimization of the tiled cholesky factorization on numa machines. In *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*. 210–217. https://doi.org/10.1109/PAAP.2012.38

[27] Wayne Joubert. 2019. Sparkler. https://github.com/wdj/sparkler.

[28] Wayne Joubert, James Nance, Sharlee Climer, Deborah A. Weighill, and Daniel A. Jacobson. 2019. Parallel accelerated Custom Correlation Coefficient calculations for genomics applications. *Parallel Comput.* 84 (2019), 15–23. https://doi.org/10.1016/J.PARCO.2019.02.003

[29] Konstantinos Kallas. 2017. Gpus-kmeans. https://github.com/angelhof/gpus-kmeans.

[30] Innovative Computing Laboratory. 2019. The High Performance LINPACK for Accelerator Introspection (HPL-AI) benchmark. https://bitbucket.org/icl/hpl-ai/src/main.

[31] Ignacio Laguna, Paul C Wood, Ranvijay Singh, and Saurabh Bagchi. 2019. Gpumixer: performance-driven floating-point tuning for gpu scientific applications. In *High Performance Computing*. Springer International Publishing, 227–246. https://doi.org/10.1007/978-3-030-20656-7_12

[32] Chris Lattner and Vikram Adve. 2004. Llvm: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[33] Guangli Li, Jingling Xue, Lei Liu, Xueying Wang, Xiu Ma, Xiao Dong, Jiansong Li, and Xiaobing Feng. 2021. Unleashing the low-precision computation potential of tensor cores on GPUs. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 90–102. https://doi.org/10.1109/CGO51591.2021.9370335

[34] Hao Lu, Michael Matheson, Vladyslav Oles, Austin Ellis, Wayne Joubert, and Feiyi Wang. 2022. Climbing the summit and pushing the

frontier of mixed precision benchmarks at extreme scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, Article 78, 15 pages. https://doi.org/10.1109/SC41404.2022.00083

[35] Zixuan Ma. 2022. A GPU FP32 computation method with Tensor Cores. https://github.com/JohndeVostok/APE.

[36] Zixuan Ma, Haojie Wang, Guanyu Feng, Chen Zhang, Lei Xie, Jiaao He, Shengqi Chen, and Jidong Zhai. 2022. Efficiently emulating high-bitwidth computation with low-bitwidth hardware. In *Proceedings of the 36th ACM International Conference on Supercomputing*. ACM, Article 5, 12 pages. https://doi.org/10.1145/3524059.3532377

[37] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *IEEE International Parallel and Distributed Processing Symposium Workshops*. 522–531. https://doi.org/10.1109/IPDPSW.2018.00091

[38] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE, Article 48, 13 pages. https://doi.org/10.5555/3291656.3291720

[39] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed precision training. In *6th International Conference on Learning Representations, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[40] Higinio Mora, María Teresa Signes-Pont, FA Pujol López, J Mora-Pascual, and JM García Chamizo. 2023. Advancements in number representation for high-precision computing. *The Journal of Supercomputing* (2023), 1–20. https://doi.org/10.1007/s11227-023-05814-y

[41] Fionn Murtagh. 1991. Multilayer perceptrons for classification and regression. *Neurocomputing* 2, 5 (1991), 183–197. https://doi.org/10.1016/0925-2312(91)90023-5

[42] Hiryuki Ootomo, Hidetaka Manabe, Kenji Harada, and Rio Yokota. 2023. Quantum circuit simulation by sdgemm emulation on tensor cores and automatic precision selection. In *High Performance Computing*, Abhinav Bhatele, Jeff Hammond, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, 259–276.

[43] Hiroyuki Ootomo, Katsuhisa Ozaki, and Rio Yokota. 2024. DGEMM on Integer Matrix Multiplication Unit. arXiv:2306.11975 [cs.DC]

[44] Hiroyuki Ootomo and Rio Yokota. 2022. Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance. *Int. J. High Perform. Comput. Appl.* 36, 4 (2022), 475–491. https://doi.org/10.1177/10943420221090256

[45] Youcef Saad and Martin H Schultz. 1986. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* 7, 3 (jul 1986), 856–869. https://doi.org/10.5555/14063.14074

[46] Huawei Technologies. 2023. HUAWEI Ascend AI Chipsets. https://www.hisilicon.com/en/products/Ascend.

[47] Pedro Valero-Lara, Ian Jorquera, Frank Lui, and Jeffrey Vetter. 2023. Mixed-Precision S/DGEMM Using the TF32 and TF64 Frameworks on Low-Precision AI Tensor Cores. ACM, 179–186. https://doi.org/10.1145/3624062.3624084

[48] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: accelerating quantized graph neural networks via GPU tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 107–119. https://doi.org/10.1145/3503221.3508408

[49] Yutong Wang and Cindy Rubio-González. 2024. Predicting Performance and Accuracy of Mixed-Precision Programs for Precision Tuning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Article 15, 13 pages. https://doi.org/10.1145/3597503.3623338

[50] Jinchen Xu, Guanghui Song, Bei Zhou, Fei Li, Jiangwei Hao, and Jie Zhao. 2024. A Holistic Approach to Automatic Mixed-Precision Code Generation and Tuning for Affine Programs. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM, 55–67. https://doi.org/10.1145/3627535.3638484

Received 2024-02-29; accepted 2024-04-01