

Article

Design and Optimization of GEMM for Complex Numbers on Ascend NPU

Erkun Zhang ^{1,2} , Yu Zhang ³, Pengxiang Xu ²  and Lu Lu ^{2,3,*} 

¹ School of Future Technology, South China University of Technology, Guangzhou 510641, China; ftcallmezk@mail.scut.edu.cn

² Pengcheng Laboratory, Shenzhen 518071, China

³ School of Computer Science & Engineering, South China University of Technology, Guangzhou 510006, China

* Correspondence: lul@scut.edu.cn

Abstract

It is widely acknowledged that General Matrix Multiplication (GEMM) serves as a foundational kernel across numerous application domains. Complex numbers exhibit distinctive mathematical properties that enable their widespread adoption across engineering computing scenarios, including signal processing and signal transformation. This study investigates high-efficiency CGEMM, namely, complex-valued GEMM, for NPU hardware, broadening the application scope of NPUs beyond mainstream low-precision AI computation workloads. The major contributions of this study are as follows: (i) numerical precision and hardware utilization of the 3M and 4M decomposition schemes on Ascend NPUs are analyzed, and the 4M method is selected as the preferred CGEMM implementation under our tested hardware constraints to fit the bandwidth limitations of modern accelerators for both precision-sensitive and performance-critical matrix computation scenarios; (ii) a complete high-performance CGEMM design based on the 4M scheme tailored for Ascend NPUs is proposed, with an AIC/AIV dual-stream pipeline scheduling strategy equipped to coordinate padding operations, matrix–matrix multiplications, and element-wise instructions across multi-level memory hierarchies and compute units; (iii) a fine-grained task scheduling and assignment mechanism is implemented to maximize Cube core occupancy across diverse matrix dimensions, improving hardware utilization for various computation workloads. Our experimental measurements show that the proposed CGEMM achieves a competitive hardware utilization rate of 83.6% across all tested matrix configurations, enabling efficient exploitation of available computing resources. Meanwhile, we observe a measured average speedup of $1.14\times$ relative to the AscendSipBoost implementation tested on an identical Ascend NPU, alongside a measured $3.17\times$ speedup compared with cuBLAS running on the Nvidia GPU platform adopted in our experiments across all evaluated matrix sizes. These results reflect the promising capability of Ascend NPUs for high-precision complex-valued computing workloads within the tested experimental setup.



Academic Editor: Paolo Bellavista

Received: 9 May 2026

Revised: 22 June 2026

Accepted: 24 June 2026

Published: 26 June 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: GEMM; ascend NPU; signal process; complex numbers

1. Introduction

It is widely acknowledged that General Matrix Multiplication (GEMM) serves as a foundational kernel across numerous application domains [1–3]. In order to obtain better parallelism so as to better exploit hardware ability, it is a common approach to bind multiple small operations into a relatively large computation, which has led to the popularity of

batched GEMM in the fields of quantum chemistry, computational fluid dynamics, tensor computations, deep learning, domain decomposition solvers, finite element methods and so on [4,5]. It is also well recognized that GEMM, as well as batched GEMM, is a keystone in versatile AI applications, including Convolution Neural Networks (CNNs) and Large Language Models (LLMs). To further accelerate big models' training, there are efforts to apply low-precision GEMMs, including those with INT8 [6], FP6 [7] and INT4 precisions [8,9].

On the other hand, complex numbers exhibit distinctive mathematical properties that enable their widespread adoption across engineering computing scenarios. For instance, the real part and imaginary part of a complex number can represent the magnitude and phase of a signal simultaneously, and operations of scaling and rotation of the signal can be realized through matrix multiplication with the complex number. Such properties of complex numbers make complex domain computations, including complex GEMM, significant to multiple fields of engineering, especially in fields of signal processing and control theory [10,11]. It is worth pointing out that complex GEMM plays a pivotal role in a variety of basic applications, including short-term Fourier transform (STFT) [10], phased-array radar (PAR), multiple-input, multiple-output (MIMO) transmission [12], orthogonal frequency division multiplexing (OFDM) transmission [13], and so on.

A Neural Processing Unit (NPU) is designed specifically for the purpose of accelerating deep learning algorithms and can be considered to be a simplified Graphics Processing Unit (GPU) that preserves the Stream Multiprocessor (SM) while removing the rendering unit [14]. This study targets the efficient implementation of standard complex-valued GEMM routines defined in BLAS libraries on Ascend NPUs, thereby extending the application scope of NPUs beyond low-precision AI workloads and unlocking their potential for high-precision complex-valued computing.

In summary, the highlights of this work are as follows:

- Numerical precision and hardware utilization of the 3M and 4M methods are analyzed for Ascend NPUs. Considering the bandwidth limitation of modern accelerators, the 4M method is adopted as the standard CGEMM implementation, which suits all matrix sizes for both precision- and performance-oriented scenarios.
- A complete design of high-performance CGEMM, or GEMM for complex numbers, using the 4M method for Ascend NPUs is presented. An AIC/AIV dual-stream pipeline scheduling scheme is developed to synchronize padding operations, matrix-matrix multiplications and element-wise instructions across hierarchical buffers and computing units.
- A fine-grained task scheduling and assignment mechanism is proposed. Tasks are sequentially dispatched to available Cube units by this mechanism. Cube-unit occupancy is maximized across diverse matrix sizes, and steady improvements in hardware utilization are achieved for various workloads.

The rest of the paper is organized as follows. Section 2 introduces the background of GEMM routines, the 3M/4M methods and the Ascend NPU architecture. Section 3 elaborates on the proposed CGEMM design by analyzing its numerical stability, performance and hardware utilization. Section 4 describes optimizations for the 4M method, including pipelining, and the fine-grained task scheduling and assignment mechanism. Section 5 evaluates the performance of the proposed CGEMM across diverse input and output matrix sizes. Section 6 concludes the paper.

2. Background

2.1. GEMM Routine

The standard $M \times N \times K$ GEMM primitive implements the fused multiply–add computation $C = \alpha AB + \beta C$. This computation relies on two scalar coefficients, α and β , alongside three dense matrices, $A_{M \times K}$, $B_{K \times N}$ and $C_{M \times N}$, requiring a total of $2MNK$ arithmetic operations. Prior to executing matrix multiplication, matrices A and B undergo panel partitioning along the M and N dimensions separately. We denote $blockC_{M_0 \times N_0}$ as one partitioned submatrix of output matrix C , computed via the product of $panelA_{M_0 \times K}$ and $panelB_{K \times N_0}$. In most practical scenarios, the K dimension spans a large range, meaning that $panelA$ and $panelB$ must be further split into j subtiles along the K axis to match the capacity of on-chip storage units. The final value of each $blockC$ is therefore computed as the cumulative sum of pairwise subtile products:

$$blockC = \sum_j panelA_j \times panelB_j \quad (1)$$

Here $panelA_j$ refers to the j -th segmented subtile extracted from $panelA$, with $panelB_j$ defined following the same rule. Independent computation tasks for distinct $blockC$ submatrices can be dispatched to separate computing cores for parallel execution.

2.2. 3M and 4M Methods

Generally speaking, there are two major methods for realizing matrix multiplication for complex numbers: the 3M method and the 4M method. Equations (2) and (3) present the computational details of the 3M method and 4M method, respectively:

$$\begin{aligned} C_r^{3M} &= C_r + A_r B_r - A_i B_i \\ C_i^{3M} &= C_i + (A_r + A_i)(B_r + B_i) - A_r B_r - A_i B_i \end{aligned} \quad (2)$$

$$\begin{aligned} C_r^{4M} &= C_r + A_r B_r - A_i B_i \\ C_i^{4M} &= C_i + A_r B_i + A_i B_r \end{aligned} \quad (3)$$

where A_r and A_i stand for the real part and imaginary part of matrix A , respectively, with a similar definition for matrices B and C [11].

Note that a practical CGEMM introduces the complex scaling factors α and β . We define five intermediate terms: $M_1 = A_r B_r$, $M_2 = A_i B_i$, $M_3 = A_r B_i$, $M_4 = A_i B_r$ and $M_5 = (A_r + A_i)(B_r + B_i)$. Let α_r and α_i denote the real and imaginary parts of the scalar α , while β_r and β_i represent the real and imaginary parts of the scalar β , respectively. Accordingly, the original formulas can be rearranged as follows:

$$\begin{aligned} C_r^{3M} &= \beta_r C_r - \beta_i C_i + \alpha_r (M_1 - M_2) - \alpha_i (M_5 - M_1 - M_2) \\ C_i^{3M} &= \beta_i C_r + \beta_r C_i + \alpha_i (M_1 - M_2) + \alpha_r (M_5 - M_1 - M_2) \end{aligned} \quad (4)$$

$$\begin{aligned} C_r^{4M} &= \beta_r C_r - \beta_i C_i + \alpha_r (M_1 - M_2) - \alpha_i (M_3 + M_4) \\ C_i^{4M} &= \beta_i C_r + \beta_r C_i + \alpha_i (M_1 - M_2) + \alpha_r (M_3 + M_4) \end{aligned} \quad (5)$$

To elaborate on their computational features, the 3M algorithm computes the resultant complex matrix C via three matrix–matrix multiplications, eight element–wise scalar multiplications and 11 element–wise additions. In comparison, the 4M algorithm requires four matrix–matrix multiplications, eight element–wise scalar multiplications and eight element–

wise additions. By reusing intermediate terms, the 3M approach reduces computational complexity and improves execution efficiency. However, repeated use of these intermediates propagates rounding errors throughout calculations and raises the risk of precision loss caused by subtraction of terms with similar magnitudes, which degrades numerical stability. On the contrary, the 4M algorithm avoids intermediate reuse and maintains fully independent multiplication paths. This design eliminates cross-propagation of rounding errors, facilitates hardware parallelization, and delivers better numerical stability.

2.3. Ascend NPU Architecture

Ascend NPUs are built to support diverse artificial intelligence workloads while delivering superior energy efficiency [15]. The hardware adopts the Single-Program–Multiple-Data (SPMD) execution paradigm: kernel routines are instantiated on all AI cores and run in parallel, and thread divergence can only be enforced by dedicated synchronization primitives. As illustrated in Figure 1, every Ascend NPU integrates two core computation components—Cube arithmetic units and Vector arithmetic units—together with dedicated special function units. Cube units are purpose-built to accelerate dense matrix multiplication, capable of finishing a complete $16 \times 16 \times 16$ matrix product in one clock cycle, equivalent to 4096 independent arithmetic calculations. By contrast, Vector units share functional similarities with GPU streaming multiprocessors, supporting element-wise computation of 256 bytes of data (128 FP32 entries) each cycle. Inter-layer data communication among all computing hardware relies on the shared off-chip L2 cache and global memory (GM) subsystem [16].

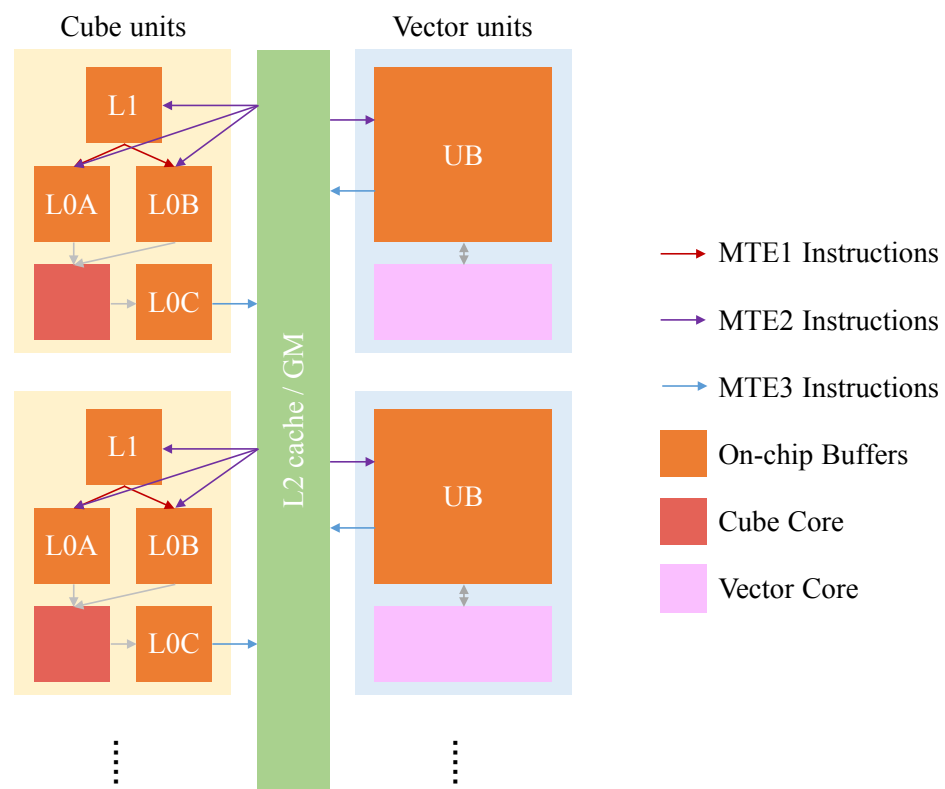


Figure 1. Architecture of an Ascend NPU and dataflows across hierarchical buffers under distinct MTE modes. The Ascend NPU consists of two kinds of computational units: Cube units for dense matrix–matrix multiplications and Vector units for element-wise operations. All computational units are connected with a shared off-chip L2 cache and global memory (GM). MTE instructions working in different modes support asynchronous execution.

Hierarchical on-chip data movement is scheduled by the Memory Transfer Engine (MTE), which adopts a minimum data access granularity of 32 bytes. Specifically, each Cube unit embeds dedicated on-chip L0 and L1 buffers, while each Vector unit is paired with an independent on-chip unified buffer (UB). The MTE provides three configurable operating modes, denoted as MTE1, MTE2, and MTE3. MTE memory access instructions in different operating modes may execute asynchronously, facilitating overlapped data transfer and arithmetic computation on Cube and Vector hardware units.

3. Design of CGEMM on Ascend NPU: 3M and 4M Method Selection

As previously discussed, both the 3M and 4M algorithms possess distinct advantages and limitations. The 4M algorithm is undoubtedly the preferred option for scenarios that cannot tolerate numerical instability. For application scenarios without stringent accuracy requirements, the 3M algorithm is expected to achieve superior performance on CPU architectures compared with the 4M algorithm, benefiting from the reduced number of matrix–matrix multiplication operations.

Nevertheless, for the Ascend NPU architecture equipped with high-performance Cube units, the 4M algorithm is likely to exhibit competitive performance. This can be inferred from the independence among its individual multiplication operations, which facilitates adequate exploitation of the computing potential of Cube units. Meanwhile, element-wise operations are bound to rely on the low-throughput Vector units and may introduce potential performance bottlenecks; therefore, the fewer element-wise instructions of the 4M algorithm render it potentially more suitable for such NPU architectures.

To determine the optimal implementation scheme of CGEMM tailored for the Ascend NPU architecture, Algorithms 1 and 2 are designed. The experimental Ascend NPU platform adopts the Huawei Atlas 300T Pro Training Card [17], an entry-level accelerator with relatively low computing capacity. Notably, both algorithms strictly adhere to the predefined computational procedures of the 3M and 4M methods, respectively, with element-wise instructions (*vadd*, *vsub* and *vmuls*) as well as the matrix–matrix multiplication instruction (*mad*) being deployed. Numerical precision and hardware utilization are analyzed in subsequent sections.

3.1. Numerical Precision Comparison

For both algorithms, numerical errors mainly stem from floating-point rounding accumulated during summation along the K dimension in matrix multiplication. On practical hardware, differences in computation scheduling and memory access arising from varying M and N also cause minor fluctuations in error results. We therefore design controlled experiments to quantitatively analyze these error characteristics.

Consistent with standard test protocols for existing CGEMM implementations, all experiments utilize matrix entries randomly sampled over $[-1, 1]$, with $\alpha = 1.0 + 0.0i$ and $\beta = 0.0 + 0.0i$, to ensure fair and consistent comparisons. This parameter combination is a common choice in prior CGEMM precision analyses, as it eliminates error terms stemming from matrix scaling and accumulation and enables direct assessment of intrinsic rounding errors induced solely by the 3M and 4M methods. Mathematically, arbitrary complex α and β only apply linear scaling to matrix operands and outputs and do not alter the relative error behavior of the two decomposition methods.

FP32 results generated on a CPU with Python 3.11 and NumPy 1.26.4 act as the golden reference. All outputs from the basic 3M and 4M CGEMM implementations discussed in this section are cross-verified against both this NumPy golden reference and cuBLAS under identical precision configurations to guarantee numerical consistency. Four error metrics are collected for quantitative evaluation: maximum relative error (MRE), maximum abso-

lute error (MAE), average relative error (ARE), and average absolute error (AAE). Each precision test is executed once, and a fixed random seed is employed to produce reproducible input matrices. Numerical errors are dominated by summation along dimension K , while dimensions M and N only introduce negligible deviations. We fix M and N to eliminate irrelevant confounding factors and isolate error trends originating from accumulation operations. Two typical configurations of output matrix C are tested to validate the generality of the observed error characteristics: a small matrix case with $M = N = 16$ and a large matrix case with $M = N = 4096$. For each configuration, M and N are kept unchanged, whereas the accumulation dimension K ranges from 1 to 32,768. The case $K = 1$ represents standalone matrix multiplication without cumulative summation, offering a precision baseline to analyze how numerical errors evolve with rising K .

Algorithm 1 Demo of the 3M method.

Input: Matrices A , B , and C in FP32 complex, scalars α and β in FP32 complex.

Output: $C = \alpha AB + \beta C$ in FP32 complex.

AIV Instructions:

- 1: Reset all sync
- 2: Load A and B from GM to UB
- 3: Separate real and imaginary parts to obtain A_r , A_i , B_r and B_i
- 4: $A_{sum} = \text{vadd}(A_r, A_i)$
- 5: $B_{sum} = \text{vadd}(B_r, B_i)$
- 6: Store A_r , A_i , A_{sum} , B_r , B_i and B_{sum} from UB to GM
- 7: Declare AIV instructions complete and jump to line 32
- 8: Load α , β and C from GM to UB
- 9: Separate real and imaginary parts to obtain α_r , α_i , β_r , β_i , C_r and C_i
- 10: $\beta_r C_r = \text{vmuls}(\beta_r, C_r)$
- 11: $\beta_i C_i = \text{vmuls}(\beta_i, C_i)$
- 12: $\beta_r C_i = \text{vmuls}(\beta_r, C_i)$
- 13: $\beta_i C_r = \text{vmuls}(\beta_i, C_r)$
- 14: $\beta C_r = \text{vsub}(\beta_r C_r, \beta_i C_i)$
- 15: $\beta C_i = \text{vadd}(\beta_r C_i, \beta_i C_r)$
- 16: Wait until AIC compute complete
- 17: Load M_0 , M_1 and M_4 from GM to UB
- 18: $M_r = \text{vsub}(M_0, M_1)$
- 19: $M_i = \text{vsub}(M_4, M_0)$
- 20: $M_i = \text{vsub}(M_i, M_1)$
- 21: $\alpha_r M_r = \text{vmuls}(\alpha_r, M_r)$
- 22: $\alpha_i M_i = \text{vmuls}(\alpha_i, M_i)$
- 23: $\alpha_r M_i = \text{vmuls}(\alpha_r, M_i)$
- 24: $\alpha_i M_r = \text{vmuls}(\alpha_i, M_r)$
- 25: $\alpha M_r = \text{vsub}(\alpha_r M_r, \alpha_i M_i)$
- 26: $\alpha M_i = \text{vadd}(\alpha_r M_i, \alpha_i M_r)$
- 27: $C_r = \text{vadd}(\alpha M_r, \beta C_r)$
- 28: $C_i = \text{vadd}(\alpha M_i, \beta C_i)$
- 29: Combine C_r and C_i to obtain target complex matrix C
- 30: Store C from UB to GM
- 31: End of CGEMM

AIC Instructions:

- 32: Wait until AIV padding complete
 - 33: Load A_r , A_i , A_{sum} , B_r , B_i and B_{sum} from GM to L1 and L0 buffers
 - 34: $M_0 = \text{mad}(A_r, B_r)$
 - 35: $M_1 = \text{mad}(A_i, B_i)$
 - 36: $M_4 = \text{mad}(A_{sum}, B_{sum})$
 - 37: Store M_0 , M_1 , M_4 from L0 buffer to GM
 - 38: Declare AIC compute complete and jump to line 16
-

Algorithm 2 Demo of the 4M method.**Input:** Matrices A , B , and C in FP32 complex, scalars α and β in FP32 complex.**Output:** $C = \alpha AB + \beta C$ in FP32 complex.**AIV Instructions:**

- 1: Reset all sync
- 2: Load A and B from GM to UB
- 3: Separate real and imaginary parts to obtain A_r , A_i , B_r and B_i
- 4: Store A_r , A_i , B_r and B_i from UB to GM
- 5: Declare AIV instructions complete and jump to line 29
- 6: Load α , β and C from GM to UB
- 7: Separate real and imaginary parts to obtain α_r , α_i , β_r , β_i , C_r and C_i
- 8: $\beta_r C_r = \text{vmuls}(\beta_r, C_r)$
- 9: $\beta_i C_i = \text{vmuls}(\beta_i, C_i)$
- 10: $\beta_r C_i = \text{vmuls}(\beta_r, C_i)$
- 11: $\beta_i C_r = \text{vmuls}(\beta_i, C_r)$
- 12: $\beta C_r = \text{vsub}(\beta_r C_r, \beta_i C_i)$
- 13: $\beta C_i = \text{vadd}(\beta_r C_i, \beta_i C_r)$
- 14: Wait until AIC compute complete
- 15: Load M_0 , M_1 , M_2 , M_3 from GM to UB
- 16: $M_r = \text{vsub}(M_0, M_1)$
- 17: $M_i = \text{vadd}(M_2, M_3)$
- 18: $\alpha_r M_r = \text{vmuls}(\alpha_r, M_r)$
- 19: $\alpha_i M_i = \text{vmuls}(\alpha_i, M_i)$
- 20: $\alpha_r M_i = \text{vmuls}(\alpha_r, M_i)$
- 21: $\alpha_i M_r = \text{vmuls}(\alpha_i, M_r)$
- 22: $\alpha M_r = \text{vsub}(\alpha_r M_r, \alpha_i M_i)$
- 23: $\alpha M_i = \text{vadd}(\alpha_r M_i, \alpha_i M_r)$
- 24: $C_r = \text{vadd}(\alpha M_r, \beta C_r)$
- 25: $C_i = \text{vadd}(\alpha M_i, \beta C_i)$
- 26: Combine C_r and C_i to obtain target complex matrix C
- 27: Store C from UB to GM
- 28: End of CGEMM

AIC Instructions:

- 29: Wait until AIV padding complete
- 30: Load A_r , A_i , B_r , B_i from GM to L1 and L0 buffers
- 31: $M_0 = \text{mad}(A_r, B_r)$
- 32: $M_1 = \text{mad}(A_i, B_i)$
- 33: $M_2 = \text{mad}(A_r, B_i)$
- 34: $M_3 = \text{mad}(A_i, B_r)$
- 35: Store M_0 , M_1 , M_2 , M_3 from L0 buffer to GM
- 36: Declare AIC compute complete and jump to line 14

As matrix elements are randomly generated within $[-1, 1]$, some output elements exhibit true values close to zero. Since MRE is highly sensitive to near-zero values and prone to evaluation distortion, we adopt MAE and AAE as the primary metrics to objectively assess numerical stability. As depicted in Figure 2, the 4M algorithm yields a slightly higher MRE than the 3M method. This discrepancy originates from extreme relative errors at individual near-zero points, rather than inferior computational accuracy. Further inspection shows that all elements with a relative error exceeding 10^{-3} have an absolute error below 10^{-5} . By comparison, the 4M algorithm achieves lower MAE and AAE, demonstrating better overall numerical stability. In addition, the AAE of the 4M method remains around 10^{-4} even when K reaches 32,768, which fully meets the precision requirements of practical engineering applications.

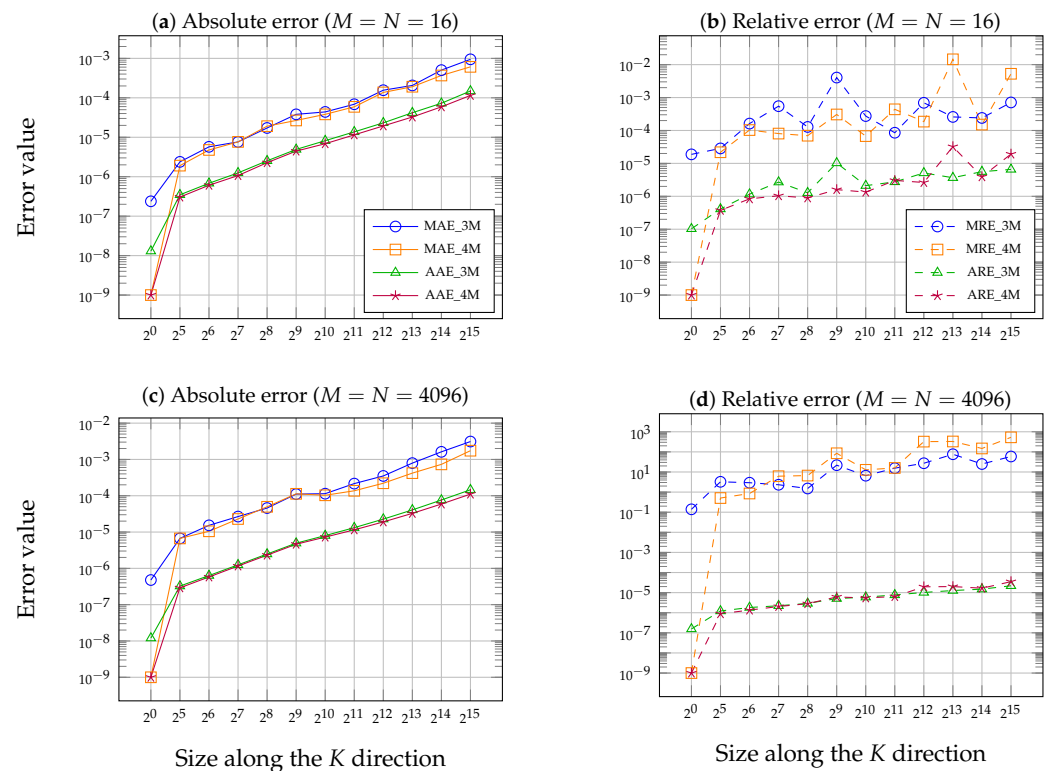


Figure 2. Comparison of error metrics between the 3M and 4M methods under various matrix configurations. Results computed using Python 3.11 and NumPy 1.26.4 on a CPU are taken as the golden reference. In the legend, MAE_3M denotes the maximum absolute error of the 3M method; all other labels follow this naming rule. Error values below 10^{-9} are represented as 10^{-9} in the figure.

3.2. Performance and Hardware Utilization Analysis

From Algorithms 1 and 2, the theoretical clock cycles under given input and output matrix sizes can be derived. We define L as theoretical clock cycles required in one CGEMM process, which should be the sum of clock cycles in matrix–matrix multiplication, element-wise matrix–scalar multiplications and element-wise accumulations. Meanwhile, E is defined as the number of Cube units in the Ascend NPU, and the number of Vector units becomes $2E$ in the real hardware configuration. Synchronization overheads and instruction initiation overheads are temporarily neglected in this stage, and L can be derived as in the following equations:

$$\begin{aligned}
 L^{3M} &= 3 \times \left\lceil \frac{2MNK}{4096E} \right\rceil + 8 \times \left\lceil \frac{MN}{128} \right\rceil + \left\lceil \frac{MK}{128} \right\rceil + \left\lceil \frac{KN}{128} \right\rceil + 9 \times \left\lceil \frac{MN}{128} \right\rceil \\
 &= 3 \times \left\lceil \frac{MNK}{2048E} \right\rceil + 17 \times \left\lceil \frac{MN}{256E} \right\rceil + \left\lceil \frac{MK}{256E} \right\rceil + \left\lceil \frac{KN}{256E} \right\rceil
 \end{aligned} \quad (6)$$

$$\begin{aligned}
 L^{4M} &= 4 \times \left\lceil \frac{2MNK}{4096E} \right\rceil + 8 \times \left\lceil \frac{MN}{128} \right\rceil + 8 \times \left\lceil \frac{MN}{128} \right\rceil \\
 &= 4 \times \left\lceil \frac{MNK}{2048E} \right\rceil + 16 \times \left\lceil \frac{MN}{256E} \right\rceil
 \end{aligned} \quad (7)$$

$$L^{4M} - L^{3M} = \left\lceil \frac{MNK}{2048E} \right\rceil - \left(\left\lceil \frac{MN}{256E} \right\rceil + \left\lceil \frac{MK}{256E} \right\rceil + \left\lceil \frac{KN}{256E} \right\rceil \right) \quad (8)$$

It can be inferred from Equation (8) that the 3M scheme yields lower theoretical clock cycles and overall floating-point computational complexity than its 4M counterpart as M , N , and K grow (i.e., $L^{4M} - L^{3M} > 0$). Nevertheless, modern AI accelerators and

general-purpose GPUs feature ample computational throughput constrained by limited off-chip memory bandwidth. On these hardware architectures, total runtime is dominated by off-chip data transfer rather than on-chip arithmetic computation. For Ascend NPUs, growing input matrix sizes inevitably degrade computational efficiency due to costly off-chip GM access, as data exchanges between Cube and Vector units rely exclusively on low-bandwidth GM.

As characterized by the classic Hockney communication model [18], off-chip data transfer time follows the linear formulation $T_{mem} = \tau + n\theta$, where τ denotes fixed startup latency, and θ represents per-byte transmission overhead proportional to data volume n .

Unlike the 4M algorithm, the 3M approach introduces extra off-chip global memory traffic for intermediate matrices $A_{sum} \in \mathbb{R}^{M \times K}$ and $B_{sum} \in \mathbb{R}^{K \times N}$, yet reduces one intermediate result of size $M \times N$ requiring off-chip data movement. Fixing synchronization overhead and core clock frequency as constants, we denote the off-chip memory bandwidth as W , which satisfies $\theta \propto W^{-1}$; the time length of one clock cycle as γ ; and the total runtime of a complete CGEMM computation as T , from which the subsequent expressions are derived:

$$T^{3M} - T^{4M} = 2\tau + 2 \times \frac{MK + NK - MN}{W} \times \text{sizeof}(\text{FP32}) - \gamma(L^{4M} - L^{3M}) \quad (9)$$

For modern AI accelerators, γ is typically negligible, so the latter term in Equation (9) can be omitted, in contrast to the costly data-movement overhead prevailing in general workloads. Accordingly, additional data transfers inevitably degrade computational efficiency on modern AI accelerators such as the Ascend NPU; consequently, the overall runtime of the 3M scheme exceeds that of its 4M counterpart for nearly all combinations of M , N and K . To further verify the accuracy of the proposed runtime model and quantify practical performance gaps across diverse matrix dimensions on physical Ascend hardware, we benchmark the two algorithms on an identical Ascend NPU platform, with various combinations of M , N , and K , in terms of runtime performance.

In a practical CGEMM deployment, padding is commonly adopted to enhance data access continuity. For performance testing, padding is omitted under the assumption that the matrix data layout fully complies with hardware storage specifications, guaranteeing contiguous memory access and consequently yielding optimal read/write performance. Each performance configuration runs 25 repeated iterations, where the first 20 iterations serve as a warm-up, and the averaged runtime of the remaining 5 runs is adopted as the final measurement. As depicted in Figure 3, the 4M method outperforms the 3M method under almost all combinations of M , N , and K , with an average acceleration of $1.18\times$. Its performance advantage becomes even more prominent as the matrix size increases. Although the 4M method brings slightly higher computational overhead, it effectively reduces data transmission over the low-bandwidth off-chip global memory (GM), allowing fuller utilization of the processor's computing resources. This aligns with a key characteristic of modern AI accelerators: the peak computing throughput is often constrained by limited off-chip bandwidth, and excessive data movement inevitably degrades overall efficiency.

We therefore select the 4M method as the preferred CGEMM implementation for our tested low-compute Ascend NPU hardware, which delivers competitive performance for both precision-sensitive and performance-critical workloads across all evaluated matrix sizes. We further speculate that the 4M method will outperform the 3M method on Ascend NPUs equipped with more computation cores without imposing substantially higher memory bandwidth capacity. In the following sections, we conduct further optimizations exclusively for the 4M-based CGEMM kernel.

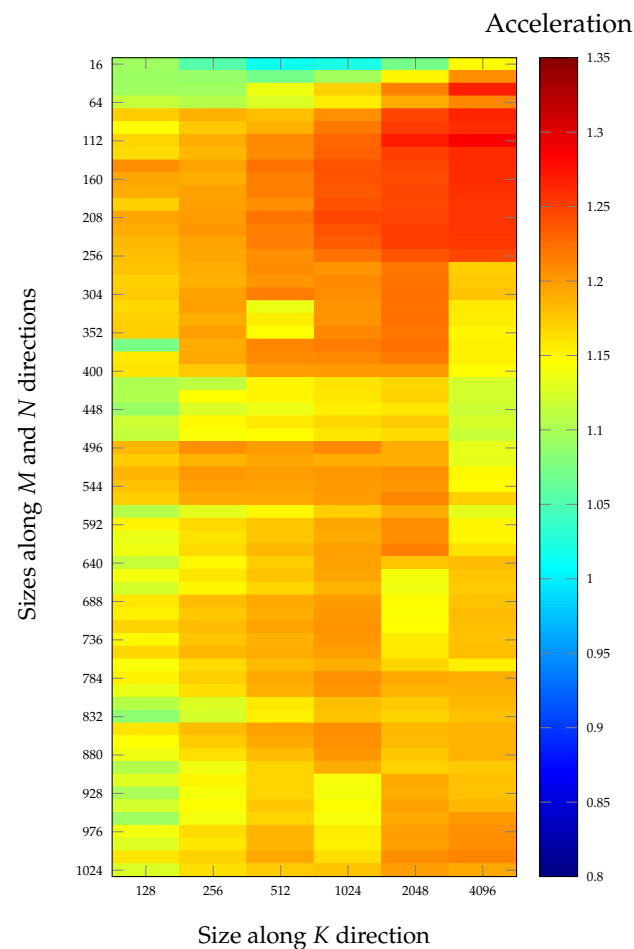


Figure 3. Acceleration of the 4M-method demo over the 3M-method demo across the joint space of M , N and K . Benefiting from cutting data transfers over low-bandwidth off-chip GM, the 4M method outperforms the 3M method under almost all matrix inputs and outputs.

4. CGEMM Optimization for Ascend NPU

4.1. Pipelining Optimizations Using Atomic Instructions

The optimized CGEMM kernel built upon the 4M method adopts an AIC/AIV dual-stream pipelining scheme across hierarchical computing hardware, enabling tight integration of MTE data transfers, Vector-based element-wise calculations, and Cube-driven matrix multiplication workloads [16]. Figure 4 presents the sequence of different operations involved in the CGEMM. Vector units first separate real and imaginary parts of complex values. Then Cube units carry out four intensive matrix–matrix multiplications to produce intermediate products. Subsequent matrix–scalar multiplication, element-wise accumulation and complex number recombination are also completed by Vector units. Furthermore, Equation (5) indicates that aggregation is required to compute the imaginary part of matrix C , which facilitates pipelining optimizations using atomic instruction-based element-wise accumulation. Atomic instructions eliminate race conditions, ensure result correctness, and avoid excessive synchronization. Without atomic operations, element-wise accumulation relies on Vector units—already tasked with complex number separation, recombination, and element-wise scalar multiplication—increasing their load and hindering overall performance optimization.

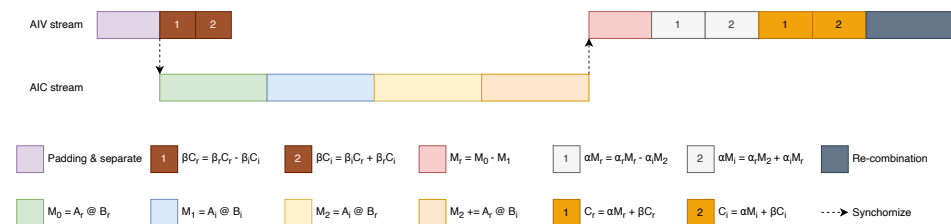


Figure 4. Timeline of the proposed 4M-CGEMM on the Ascend NPU, adopting the AIC/AIV dual-stream pipeline and atomic operation optimizations. The AIV stream executes element-wise operations on Vector units, and the AIC stream carries out dense matrix–matrix multiplications on Cube units.

4.2. Fine-Grained Task Scheduling and Assignment

In fields of communication as well as signal processing, the matrix shape in CGEMM significantly varies between different applications. Usually, M , N and K lie between 16 and 1024 in applications of STFT [10] and PAR, between 1024 and 4096 in an application of MIMO transmission [12], and between 4096 and 32,768 in an application of OFDM transmission [13].

To maximize the occupancy of Cube units across diverse input and output matrix sizes, especially when M and N directions are small, we propose a fine-grained task scheduling and assignment mechanism for matrix–matrix multiplication, where a task refers to a single Cube unit computing one specified block from matrix M_0 , M_1 , M_2 or M_3 . Inspired by batched GEMM, which coalesces multiple lightweight operations into a compute-intensive kernel [19–21], this mechanism enables sequential task dispatching to available Cube units. The example depicted in Figure 5 better demonstrates the proposed mechanism: each matrix–matrix multiplication is partitioned into 16 blocks, generating 64 tasks for the 4M-CGEMM kernel, with 22 Cube units available on the hardware. Without the proposed mechanism, tasks are assigned by Block Index, taking 4 computation rounds in total, with 16 tasks processed per round and 6 Cube units remaining idle. In contrast, the proposed mechanism assigns tasks by Mission Index, reducing the total rounds to 3. This reduces the total computation rounds by one and keeps nearly all Cube units fully occupied for better hardware utilization. Beyond this case, the proposed mechanism is applicable to matrix sizes of all scales and consistently improves hardware occupancy across various workloads.

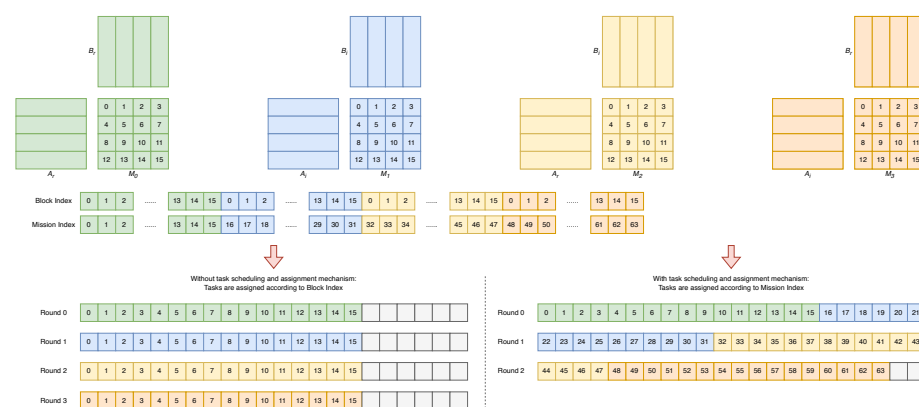


Figure 5. An example of the proposed task scheduling and assignment mechanism. In this example, one matrix–matrix multiplication is divided into 16 blocks, and the hardware is equipped with 22 available Cube units. A task refers to a single Cube unit computing one specified block from matrix M_0 , M_1 , M_2 or M_3 . Without the proposed mechanism, tasks are assigned by Block Index. This scheme takes 4 computation rounds, with 16 tasks processed per round and 6 Cube units remaining idle. By contrast, when adopting the proposed mechanism, tasks are assigned by Mission Index. Only 3 rounds are required, reducing the total computation rounds by one and improving hardware utilization.

5. Evaluation

In this section, our proposed CGEMM is evaluated. The Huawei Atlas 300T Pro Training Card [17] is selected as the Ascend NPU environment, while an Nvidia A800 GPU [22] is selected for cross-platform comparison. We first evaluate the effectiveness of pipelining optimizations using atomic instructions, then assess the effectiveness of the task assignment mechanism, and finally compare the performance of the proposed CGEMM with that of asdBlasCgemm from AscendSiPBoost [23] on an identical Ascend NPU, as well as with that of the cuBLAS implementation using default mode on an Nvidia GPU. Note that Ampere GPUs do not enable Tensor Core for complex GEMM. The default mode prioritizes performance via relaxed floating-point operations, whereas the pedantic mode enforces strict IEEE 754 compliance and fixed computation order. Table 1 gives a more detailed description of implementation details. The value of each element in matrices A , B and C , as well as constants α and β , is randomly generated between 0 and 1 using a fixed random seed. For each run, execution is repeated 25 times, where the first 20 times serve as a warm-up, and the average execution time of the last 5 times is taken as the result. With consideration that a single GEMM accomplishes $2MNK$ arithmetic operations in total, the performance metric TFLOPS of CGEMM can be calculated using the following equation:

$$\text{TFLOPS} = \omega \times \frac{2 \times M \times N \times K}{T \times 10^{12}} \quad (10)$$

where M , N , and K denote the dimensions of the matrices, T is the average execution time of the last 5 runs (in seconds) of a single CGEMM, and ω is a coefficient: $\omega = 3$ for the 3M method, and $\omega = 4$ for the 4M method.

Table 1. Implementation details for CGEMM performance evaluation.

	Huawei Atlas 300T Pro	Nvidia A800
Buffers	L1 512 KB (per core)	L1 192 KB (per SM)
Caches	L2 192 MB	L2 40 MB
Memory	64 GB HBM2e	80 GB HBM2e
Memory Bandwidth	1.6 TB/s	2.04 TB/s
FP32 Peak Throughput	70 TFLOPS	19.5 ¹ /156 ² TFLOPS
Programming Language	Ascend C	C++
Compilers	Bisheng, C++17	NVCC
Library & Runtime API	CANN 8.0.0	cuBLAS 12.6
GEMM implementation	(1) Proposed CGEMM (2) asdBlasCgemm	cublasCgemm (default) ^{3,4}
Host Environment	ARM-based Kunpeng CPU Python 3.11 NumPy 1.26.4	ARM-based CPU Python 3.11 NumPy 1.26.4

¹ Peak FP32 throughput of CUDA Core for complex GEMM. ² Tensor Core TF32 peak throughput, not available for complex GEMM. ³ Tensor Core does not support complex GEMM on Ampere architecture. ⁴ The default math mode optimizes for performance, and the pedantic mode ensures strict IEEE 754 compliance and numerical precision.

5.1. Effectiveness of Pipelining Optimizations Through Atomic Instructions

To validate the practical value of the proposed pipelining optimizations, we first evaluate their effectiveness across a broad range of input and output matrix sizes. Specifically, we quantify the performance improvement by comparing the execution durations of CGEMM implementations with and without atomic instructions. As illustrated in Figure 6, atomic instructions successfully mitigate synchronization overhead, achieving a maximum acceleration of $1.05\times$ and an average acceleration of $1.02\times$. These results verify the efficacy

of pipelining optimizations in reducing synchronization overhead, providing a dependable baseline for exploring further hardware-aware computation optimizations.

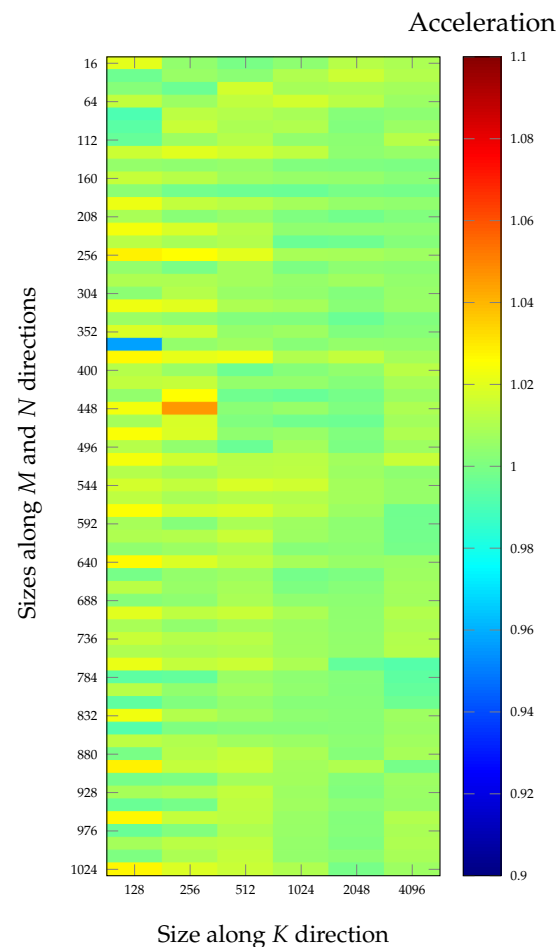


Figure 6. Effectiveness of pipelining optimizations through atomic instructions under different sizes of matrix inputs and outputs.

5.2. Effectiveness of Fine-Grained Task Scheduling and Assignment

Having validated the effectiveness of pipelining optimizations, we proceed to evaluate the proposed fine-grained task scheduling and assignment mechanism. We quantify the performance gains by comparing the execution latency of the CGEMM implementations with and without the proposed mechanism, where atomic instructions are disabled for all experimental tests. As illustrated in Figure 7, the proposed mechanism effectively alleviates the low Cube-unit occupancy issue—a key bottleneck limiting computational efficiency—for matrices with small M and N dimensions. This optimization yields a maximum speedup of $2.18\times$ for small matrices and an average speedup of $1.32\times$ across diverse matrix configurations. The results verify that the proposed task scheduling and assignment mechanism is applicable to matrix sizes of all scales and achieves consistent improvements in hardware occupancy across various workloads.

5.3. CGEMM Performance Evaluation

With both block-wise pipelining optimizations and the proposed task assignment mechanism fully enabled, we conduct comprehensive performance evaluations of our CGEMM implementation over a wide spectrum of matrix shapes to demonstrate its robustness and practical applicability. Nevertheless, CGEMM merely serves as a fundamental kernel within STFT, PAR, MIMO and OFDM systems. Such real-world applications integrate

numerous compute-intensive primitives and intricate mathematical pipelines. The performance evaluation of this work centers on general BLAS-level characteristics, and end-to-end assessments targeting complete application workflows lie beyond the research scope and technical capacity of the current study.

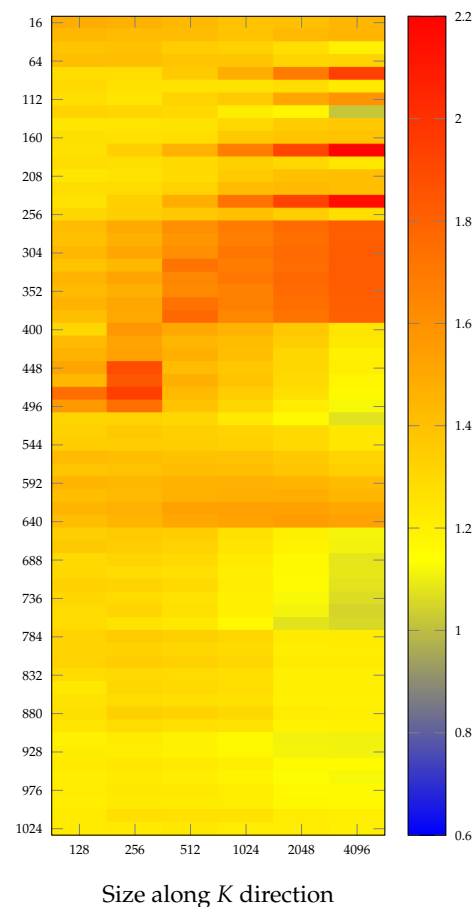


Figure 7. Effectiveness of the proposed task scheduling and assignment mechanism under different sizes of matrix inputs and outputs.

Given the distinct gap in FP32 peak computability between the Ascend NPU and Nvidia GPU, we evaluate both TFLOPS and hardware utilization. As listed in Table 1, the peak computability is 70 TFLOPS for the Ascend NPU and 19.5 TFLOPS for the Nvidia GPU, as Tensor Core acceleration is unavailable for complex GEMM on the Ampere architecture. In total, 2500 synthetic CGEMM cases are generated, where M , N and K are randomly sampled between 256 and 32,768 to reflect typical communication and signal processing scenarios and directly applied to all implementations involved on the Ascend NPU and Nvidia GPU.

As depicted in Figure 8, the CGEMM implementation from AscendSipBoost achieves a peak performance comparable to that of our proposed method. Meanwhile, our CGEMM delivers a steady average speedup of $1.14\times$ over `asdBlasCgemm` with considerably lower performance fluctuation—an essential advantage for stable and reliable high-performance computing. More notably, under the hardware and software configurations listed in Table 1, our CGEMM implementation achieves a measured $3.17\times$ speedup compared with cuBLAS, demonstrating favorable hardware resource utilization efficiency for complex matrix multiplication under these test conditions. The relative standard deviation of all measured TFLOPS values is below 1%, with absolute measurement fluctuations on the order of 10^{-4} or smaller. This minimal variance renders error bars visually indiscernible in

plotted curves; consequently, only average throughput values are presented throughout all figures.

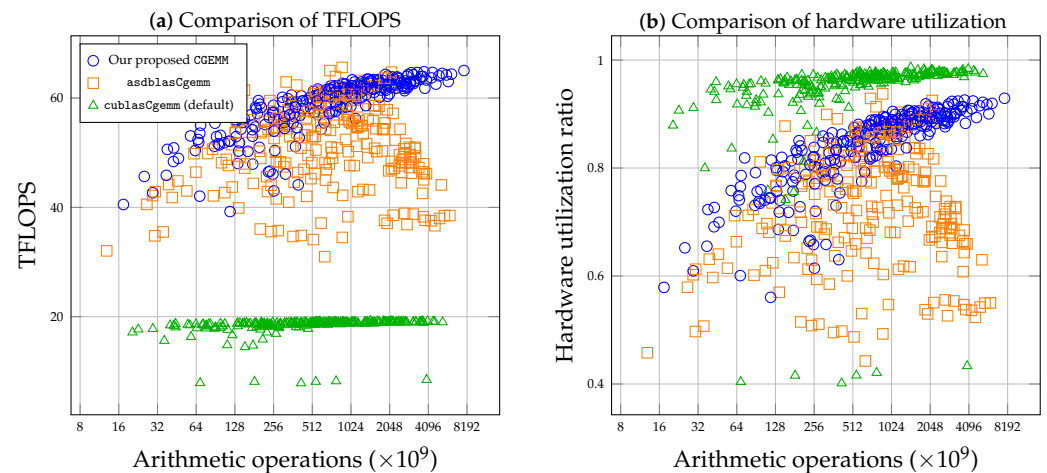


Figure 8. Comparison of performance and hardware utilization of CGEMM from different implementations on the Ascend NPU and Nvidia GPU across diverse matrix inputs and outputs. Throughout the experiment, cublasCGemM is set to the default mode, which prioritizes performance by adopting relaxed floating-point arithmetic rules.

In terms of hardware utilization, cuBLAS attains an average utilization rate of 94.5%. Despite this gap, our implementation still achieves a satisfactory 83.6% hardware utilization under diverse matrix configurations, maintaining efficient use of the underlying computing resources.

6. Conclusions

As one of the most widely used high-performance kernels, GEMM plays a keystone role in various application fields. In particular, complex GEMM is indispensable for handling complex-valued matrix operations, which are ubiquitous in these fields but remain under-optimized for NPU hardware—traditionally tailored for low-precision AI-related computations. This study fills this critical gap by exploring and optimizing efficient CGEMM implementations on an Ascend NPU, thereby extending the application scope of NPUs beyond conventional AI tasks and unlocking their potential for high-precision complex-valued computing. By adopting the 4M method—our preferred approach, validated on the tested Ascend NPU hardware architecture—together with pipelining optimizations and the fine-grained task scheduling and assignment mechanism, our proposed CGEMM achieves a measured average speedup of $1.14\times$ over the AscendSipBoost implementation on the same tested Ascend NPU and a $3.17\times$ measured speedup over the cuBLAS implementation on the Nvidia GPU used in our experiments across all evaluated matrix sizes. It also delivers a satisfactory hardware utilization rate of 83.6% under diverse matrix configurations, enabling efficient usage of the underlying computing resources. These results not only validate the effectiveness of our proposed optimizations but also lay a solid foundation for fully exploiting the computing power of Ascend NPUs in complex-valued computing scenarios, providing valuable technical support for advancing related research and industrial applications.

Author Contributions: Conceptualization, E.Z. and L.L.; methodology, E.Z.; validation, E.Z., P.X. and L.L.; formal analysis, E.Z.; writing—original draft preparation, E.Z.; writing—review and editing, Y.Z., P.X. and L.L.; visualization, E.Z.; supervision, L.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Acknowledgments: The Ascend NPU hardware experimental environment is supported in part by Guangdong S&T Program (2024B0101010003).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AIC	AI Cube Core
AIV	AI Vector Core
CPU	Central Processing Unit
GEMM	General Matrix Multiplication
GM	Global Memory
MIMO	Multiple-Input, Multiple-Output
NPU	Neural Processing Unit
MTE	Memory Transfer Engine
OFDM	Orthogonal Frequency Division Multiplexing
PAR	Phased-Array Radar
SM	Stream Multiprocessor
SPMD	Single-Program-Multiple-Data
STFT	Short-Term Fourier Transformation
TFLOPS	Tera Floating-point Operations Per Second
UB	Unified Buffer

References

1. Cicek, N.M.; Shen, X.; Ozturk, O. Energy Efficient Boosting of GEMM Accelerators for DNN via Reuse. *ACM Trans. Des. Autom. Electron. Syst.* **2022**, *27*, 1–26. [\[CrossRef\]](#)
2. Drmač, Z. A LAPACK Implementation of the Dynamic Mode Decomposition. *ACM Trans. Math. Softw.* **2024**, *50*, 1–32. [\[CrossRef\]](#)
3. Nair, H.; Vellaisamy, P.; Chen, A.; Finn, J.; Li, A.; Trivedi, M.; Shen, J.P. tuGEMM: Area-Power-Efficient Temporal Unary GEMM Architecture for Low-Precision Edge AI. In Proceedings of the 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 21–25 May 2023; pp. 1–5. [\[CrossRef\]](#)
4. Jiang, L.; Yang, C.; Ma, W. Enabling Highly Efficient Batched Matrix Multiplications on SW26010 Many-core Processor. *ACM Trans. Archit. Code Optim.* **2020**, *17*, 1–23. [\[CrossRef\]](#)
5. Mijić, N.; Davidović, D. Batched matrix operations on distributed GPUs with application in theoretical physics. In Proceedings of the 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 23–27 May 2022; pp. 293–299. [\[CrossRef\]](#)
6. Dettmers, T.; Lewis, M.; Belkada, Y.; Zettlemoyer, L. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Proceedings of the Advances in Neural Information Processing Systems*; Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2022; Volume 35, pp. 30318–30332.
7. Xia, H.J.; Zheng, Z.; Wu, X.X.; Chen, S.Y.; Yao, Z.W.; Youn, S.; Bakhtiari, A.; Wyatt, M.; Zhuang, D.L.; Zhou, Z.Z.; et al. Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs. In Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24), Santa Clara, CA, USA, 10–12 July 2024; pp. 699–713.
8. AMD. rocBLAS 5.2.0 Documentation. Available online: <https://rocm.docs.amd.com/projects/rocBLAS/en/latest> (accessed on 23 June 2026).
9. Wu, X.; Li, C.; Yazdani Aminabadi, R.; Yao, Z.; He, Y. Understanding Int4 Quantization for Language Models: Latency Speedup, Composability, and Failure Cases. In *Proceedings of the 40th International Conference on Machine Learning*; Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J., Eds.; Proceedings of Machine Learning Research; PMLR: Cambridge, MA, USA, 2023; Volume 202, pp. 37524–37539.
10. Yoneyama, R.; Kawamura, M.; Terashima, R.; Yamamoto, R.; Toda, T. Comparative Analysis of Fast and High-Fidelity Neural Vocoder for Low-Latency Streaming Synthesis in Resource-Constrained Environments. *arXiv* **2025**, arXiv:2506.03554.

11. Van Zee, F.G.; Smith, T.M. Implementing High-performance Complex Matrix Multiplication via the 3m and 4m Methods. *ACM Trans. Math. Softw.* **2017**, *44*, 1–36. [\[CrossRef\]](#)
12. Hassan, M.W.; Dabah, A.; Ltaief, H.; Fahmy, S.A. Signal Detection for Large MIMO Systems Using Sphere Decoding on FPGAs. In Proceedings of the 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), St. Petersburg, FL, USA, 15–19 May 2023; pp. 102–111. [\[CrossRef\]](#)
13. Lin, Q.; Shen, H.; Li, Z.; Xu, W.; Zhao, C.; You, X. One-Bit Transceiver Optimization for mmWave Integrated Sensing and Communication Systems. *IEEE Trans. Commun.* **2025**, *73*, 800–816. [\[CrossRef\]](#)
14. Lee, K.J. Chapter Seven - Architecture of neural processing unit for deep neural networks. In *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*; Advances in Computers; Kim, S., Deka, G.C., Eds.; Elsevier: Amsterdam, The Netherlands, 2021; Volume 122, pp. 217–245. [\[CrossRef\]](#)
15. Liao, H.; Tu, J.; Xia, J.; Liu, H.; Zhou, X.; Yuan, H.; Zhang, L.; Wang, Y.; Li, W.; Chen, K. Ascend: A Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing: Industry Track Paper. In Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Republic of Korea, 27 February–3 March 2021; pp. 789–801. [\[CrossRef\]](#)
16. Zhang, E.; Xu, P.; Lu, L. Efficient Low-Precision GEMM on Ascend NPU: HGEMM’s Synergy of Pipeline Scheduling, Tiling, and Memory Optimization. *Computers* **2026**, *15*, 39. [\[CrossRef\]](#)
17. Huawei. Atlas 300T Pro Training Card User Guide (Model 9000) 06. Available online: <https://support.huawei.com/enterprise/en/doc/EDOC1100285917/426cfd9/about-this-document?idPath=23710424|251366513|22892968|252309113|256118195> (accessed on 1 June 2026).
18. Hockney, R.W. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* **1994**, *20*, 389–398. [\[CrossRef\]](#)
19. Abdelfattah, A.; Haidar, A.; Tomov, S.; Dongarra, J. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *International Conference on High Performance Computing*; Kunkel, J.M., Balaji, P., Dongarra, J., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 21–38.
20. Dongarra, J.; Hammarling, S.; Higham, N.J.; Relton, S.D.; Valero-Lara, P.; Zounon, M. The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems. *Procedia Comput. Sci.* **2017**, *108*, 495–504. [\[CrossRef\]](#)
21. Abdelfattah, A.; Costa, T.; Dongarra, J.; Gates, M.; Haidar, A.; Hammarling, S.; Higham, N.J.; Kurzak, J.; Luszczek, P.; Tomov, S.; et al. A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines. *ACM Trans. Math. Softw.* **2021**, *47*, 1–23. [\[CrossRef\]](#)
22. Nvidia. NVIDIA A800 80GB Active Graphics Card. Available online: <https://www.nvidia.com/en-us/products/workstations/a800> (accessed on 24 October 2023).
23. Huawei. Domain-Specific Libraries. Available online: <https://www.hiascend.com/eng/cann/aal?tab=1> (accessed on 23 June 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.