Unlocking High Performance with Low-Bit NPUs and CPUs for Highly Optimized HPL-MxP on Cloud Brain II

Weicheng Xue
Pengcheng Laboratory
Shenzhen, Guangdong, China 518055
xuewch@pcl.ac.cn

Kai Yang
Pengcheng Laboratory
Shenzhen, Guangdong, China 518055
yangk@pcl.ac.cn

Yongxiang Liu
Pengcheng Laboratory
Shenzhen, Guangdong, China 518055
liuyx@pcl.ac.cn

Dengdong Fan
Pengcheng Laboratory
Shenzhen, Guangdong, China 518055
fandd@pcl.ac.cn

Pengxiang Xu
Pengcheng Laboratory
Shenzhen, Guangdong, China 518055
xupx@pcl.ac.cn

Yonghong Tian
Pengcheng Laboratory
Shenzhen, Guangdong, China 518055
tianyh@pcl.ac.cn

***Abstract*—Mix-precision computation is crucial for artificial intelligence and scientific computing applications. However, as novel chips with innovative architectures emerge, harnessing their computational capabilities presents significant challenges. While existing algorithms for the HPL-MxP LU factorization excel on homogeneous systems, they often encounter difficulties on specialized heterogeneous architectures. This deficiency arises from inadequate optimization for computation, memory access, and communication, hindering effective mixed-precision acceleration. This work introduces an algorithm-hardware co-optimization approach for LU factorization on specialized NPUs and CPUs, leveraging their unique architectures. A novel multi-iteration fusion method for general matrix multiplication is proposed, strategically designed to maximize on-chip L1 buffer utilization, effectively overcoming the notorious "memory wall". Additionally, a multi-stage, multi-level heterogeneous pipeline for LU factorization in an accelerator-CPU cloud environment is presented, where compute-intensive matrix multiplications are offloaded to NPUs while CPUs handle the remaining tasks. The co-optimization approach fosters deep collaboration between CPUs and accelerators, thereby unlocking enhanced performance.**

I. Introduction

The HPL-MxP benchmark [1], [2] offers critical insights for evaluating and optimizing the performance of large-scale matrix computations on emerging artificial intelligence (AI) chips. Acting as a pivotal bridge between high-performance computing (HPC) and artificial intelligence, the benchmark employs low-precision LU factorization followed by generalized minimal residual method (GMRES) iteration to reinstate high-precision numerical accuracy [3]–[6]. This benchmark underscores the convergence of both computational speed and accuracy in modern AI chips tailored for AI workloads, thus drawing considerable interest as a powerful method for evaluating innovative AI chip designs.

AI chips, encompassing various types such as graphic processing units (GPUs), tensor processing units (TPUs), neural processing units (NPUs), field programmable gate arrays (FPGAs), and application specific integrated circuits (ASICs), are specialized compute units designed to accelerate diverse artificial intelligence tasks. Each of these AI accelerators has unique architectures and designs, tailored for various workloads. GPUs, with their parallel processing units and high memory bandwidth, excel in general-purpose parallel computing tasks like deep learning and scientific computing [7]. TPUs leverage specialized tensor processing units in systolic arrays for tensor acceleration in deep learning [8], while NPUs utilize specialized matrix computation units for efficient large-scale matrix operations essential for neural network computations [9]. FPGAs and ASICs can offer high performance and energy efficiency for highly customized computing tasks such as deep learning inference acceleration [10]. Overall, these AI chips play a crucial role in high performance AI computing, leveraging specialized hardware architectures for efficient tensor operations and low-precision calculations.

The burgeoning demand for computing power from AI chips for both training and inference tasks has escalated with the development of large-scale models [11]–[13]. However, this growth in computing power has far outpaced the read-write bandwidth of chip memory in recent years, also known as the "memory wall" [14]–[16]. For instance, consider the widely used Nvidia GPU: its computing power has surged by a factor of 1000 (from 21 TFlops in FP16 for P100 to 20 PFlops in Tensor FP4 for B200) over the past 8 years, while its memory bandwidth has only increased tenfold (from 732 GB/s for P100 to 8 TB/s for B200) during the same period. This disparity presents a significant obstacle to fully harnessing the computing potential of AI chips, since memory transfer emerges as the main limiting factor in AI applications, especially noticeable in large language models [16]. Addressing this "memory wall" challenge necessitates a rethink of optimizing memory access for application algorithms and efficiently leveraging the heterogeneous computing power of chips in alignment with

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

their specific hardware architectures.

Performing computations on novel specialized architectures presents significant challenges when relying on open-source libraries. A key obstacle is the lack of support for heterogeneity, inhibiting the efficient utilization of hardware heterogeneity. While some prior works have addressed portability or heterogeneity [17], [18], certain low-level optimizations remain necessary to fully exploit the compute capabilities of novel AI chips. Moreover, the integration of mixed-precision computing further complicates this issue, as low-bit precision may lead to potential overflows, underflows, or numerical stability issues. In heterogeneous environments with mixed-precisions, the algorithm-level mixed-precision computing are closely related to hardware-level heterogeneous computing components, such as low-bit AI chips and high-bit CPUs. This approach optimizes both computational speed and result accuracy by leveraging numerical values of varying precisions provided by heterogeneous computing components during computation [19]–[21]. The selection and configuration of heterogeneous computing components impact both the effectiveness and performance of mixed-precision computing. Consequently, a second challenge ("mixed-precision optimization") arises: how to carefully select appropriate computing units for computational tasks with varying precision requirements, thereby harnessing heterogeneous computing resources while preserving algorithmic accuracy.

Fully leveraging computational resources in heterogeneous environments to unlock high performance involves carefully considering various factors such as understanding system hardware characteristics, managing data transfer and synchronization, and balancing workloads. Designing efficient heterogeneous acceleration pipelines on novel hardware architectures poses a significant challenge ("heterogeneous pipeline optimization"), as it involves maximizing the combined benefits of different computing resources to improve overall efficiency and throughput. Prior works have explored the use of pipelines to enhance compute throughput in diverse systems, such as recommendation systems [22], [23], CPU-GPU heterogeneous environments with the Linpack algorithm [24], and graph processing systems on FPGAs [25].

To overcome these challenges, it is crucial to optimize computation, memory access, and communication on novel heterogeneous platforms, as well as to improve the efficiency of mixed-precision calculations and design effective heterogeneous computing pipelines. These efforts are essential for fully realizing the high performance potential of novel AI chips. In this work, we make the following contributions:

- We propose a novel multi-iteration fusion method for general matrix multiplication (GEMM) to efficiently address the "memory wall" challenge. This approach effectively reduces data transfer from global memory and maximizes the utilization of the low-latency and high-bandwidth on-chip L1 buffer on the NPU.
- We propose a co-acceleration method for triangular solve with matrix (TRSM) using both the CPU and NPU to effectively tackle the "mixed-precision optimization" challenge. This involves offloading low-cost small matrix inversion calculations in TRSM to the CPU using FP32, while assigning matrix multiplication in TRSM to the NPU using FP16.
- We propose a novel multi-stage, multi-level heterogeneous pipeline for LU factorization on the CPU and NPU, addressing the "heterogeneous pipeline optimization" challenge. By performing only GEMM on the NPU and the rest on the CPU, this optimization significantly enhances the performance. Overall, this method integrates the multi-iteration fusion method and the co-acceleration method for TRSM.

While our work is focused on the NPU environment, we believe that our proposed methods can be readily adapted to other heterogeneous computing systems utilizing AI chips with different architectures, as these methods incorporate optimizations at multiple layers to improve performance.

II. HPL-MxP and Related Works

A. HPL-MxP

Traditional high-performance computing primarily focuses on fields like physics, chemistry, and biology, requiring double precision calculations. However, the rise of artificial intelligence has led to a paradigm shift in computational requirements, where AI techniques have demonstrated efficacy in lower precision formats, such as single precision or even lower. Consequently, modern supercomputers are increasingly employed for AI applications, benefiting from the efficiency and cost-effectiveness of low-precision operations. Traditional benchmarks like HPL [26] may not adequately evaluate the performance of these AI-oriented supercomputers, prompting the development of HPL-MxP [1]. This novel approach combines LU decomposition in low precision with subsequent precision iterations to restore solutions to 64-bit precision, addressing the evolving computational demands of AI workloads.

The core operation of HPL and HPL-MxP revolves around block LU factorization, a critical step in solving linear equations, depicted in Fig. 1. This process involves decomposing a matrix into lower triangular and upper triangular matrices, achieved through a series of operations including diagonal updates, triangular matrix solves with matrix right-hand sides (TRSM), and general matrix multiplication (GEMM). These operations are iteratively applied to the trailing submatrix until the entire factorization is solved. The LU factorization is executed through the following procedures:

1) Diagonal update: At iteration step $i-1$, perform diagonal update on matrix block A_{ii} to compute lower-triangular L_{ii} and upper-triangular U_{ii}, where $A_{ii} = L_{ii}U_{ii}$.
2) Triangular solve with matrix right hand sides (TRSM): perform TRSM with inputs U_{ii} and A_{ji} to compute the left panel L_{ji}, where $A_{ji} = L_{ji}U_{ii}$ for $j \geq i+1$.
3) Triangular solve with matrix right hand sides (TRSM): perform TRSM with inputs L_{ii} and A_{ij} to compute the right panel U_{ij}, where $A_{ij} = L_{ii}U_{ij}$ for $j \geq i+1$.
4) GEMM update (trailing matrix update): Perform GEMM with inputs L_{ik}, U_{kj} obtained from steps 2 and 3, and A_{jj}

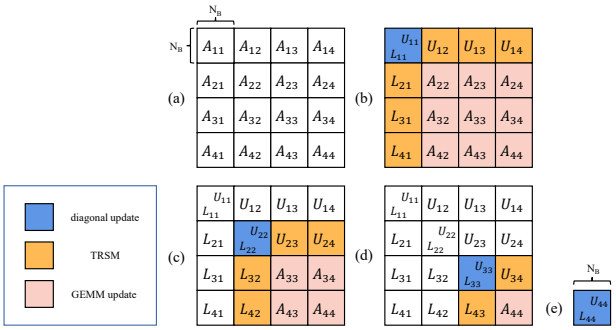

Fig. 1. Block LU factorization

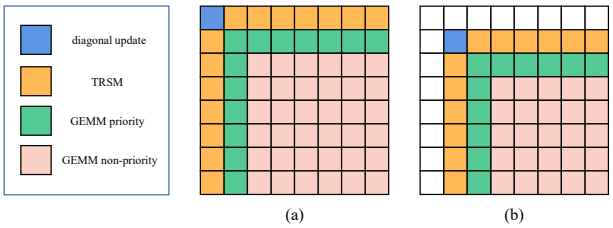

Fig. 2. Look-ahead in LU factorization

to update the submatrix A_{jj} for the next iteration, where $A_{ij\text{new}} = A_{ij} - L_{ik}U_{kj}$. A_{ij} represents all the remaining submatrix blocks.

5) Repeat procedures 1-4 to factorize the leftmost column and top row matrix blocks, updating the remaining submatrices until all matrix blocks are factorized.

An established optimization technique in LU factorization is the look-ahead method [27]. It divides the trailing submatrix into two distinct zones: the GEMM priority zone and the GEMM non-priority zone, seen in Fig. 2. The GEMM priority area must be completed before the next iteration of diagonal update and TRSM, which can proceed without waiting for the GEMM non-priority area from the current iteration. Figures 2 (a) and (b) illustrate the initial and subsequent iteration steps, respectively. Notably, the diagonal update and TRSM depicted in Fig.2 (b) critically rely on the computation of the GEMM priority zone shown in Fig.2 (a), emphasizing the necessity for prioritized computation of the GEMM priority zone.

B. Related Works

The HPL-MxP (formerly HPL-AI) running on the Fugaku supercomputer marked a significant milestone by achieving Exascale performance [28]. This achievement was made possible through the implementation of multiple optimization techniques aimed at achieving high parallel efficiency across a large number of homogeneous nodes. The Fugaku supercomputer comprises over 100,000 A64FX CPUs based on the ARMv8.2 architecture. Additionally, the HPL-MxP implementation on two of the world's fastest supercomputers, Summit and Frontier, demonstrated scalability at extreme scale [27]. These systems utilize general purpose GPUs for mixed-precision acceleration, albeit without employing data pipelining between the CPU and GPU during factorization. Their approach aligns with the goal of maximizing execution on GPUs due to their superior performance and larger high-bandwidth memory capacity over CPUs, a distinction from our work. Guidelines for optimizing strategies are shared to achieve portable performance across various GPUs at an extreme scale. Notably, the Sunway supercomputer achieved 5 ExaFlop/s with 85% parallel efficiency and linear scalability across its 40 million cores, leveraging optimization methods tailored to the system, with a focus on network topology and bandwidth pruning for large-scale supercomputer facilities.

Different from Fugaku and Sunway using homogeneous CPUs, and Summit and Frontier using heterogeneous CPUs and GPUs, the Cloud Brain II AI supercomputer adopts totally different architectures with heterogeneous Huawei Ascend NPUs and Kunpeng CPUs. The Ascend NPU is tailored for neural network computations, boasting powerful low-bit computing capabilities. However, its versatility falls short of GPUs, lacking extensive single-precision computing capabilities, which is common in novel AI chips. Therefore, well-designed heterogeneous algorithms are necessary to enhance the performance of HPL-MxP.

C. Existing Implementation Techniques

All existing techniques mentioned have already been established and should not be considered novel contributions of this work. However, they serve as important components in our implementation and optimization of the HPL-MxP benchmark on Cloud Brain II, building on best practices and insights from Kudo et al. [28] and Lu et al. [27].

1) Single Iteration Scheme for IR: The single-iteration scheme for iterative refinement (SIIR) is a simplified version of the iterative refinement method used in the HPL-MxP implementation on Fugaku [28]. It accelerates convergence by employing a carefully chosen initial guess, achieving faster and more efficient convergence with one iteration. SIIR offers significant improvements in accuracy and computational efficiency, making it ideal for the HPL-MxP benchmark.

2) Lazy Initialization: Lazy initialization involves rearranging the summation order so that the initial value of the matrix typically having the largest magnitude is computed last [28].

3) On-The-Fly Matrix Generation: On-the-fly matrix generation reduces memory usage by dynamically creating matrices as needed, eliminating the need to store large FP64 matrices in memory [28]. This technique enhances the efficiency of iterative refinement, resulting in reduced memory overhead and improved computational performance.

4) Look-Ahead: Look-ahead optimization postpones the full Update Trailing Matrix phase to the next iteration, enabling overlap of communication and computation, thus improving performance [27]. As HPL-MxP can be memory bound, balancing these overlapped terms is crucial for optimizing performance in memory bound scenarios.

5) 2D Cyclic Decomposition: All processes are organized into a two-dimensional process grid, where the input matrix

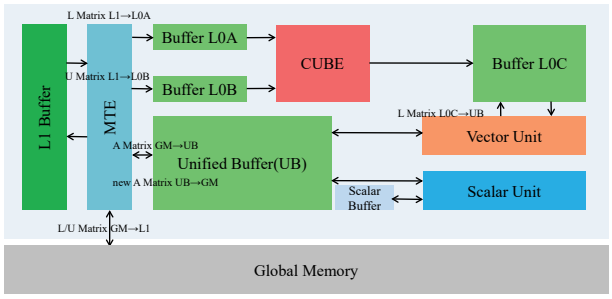

Fig. 3. Architecture of the Huawei Ascend NPU AI core

A is mapped in $N_B \times N_B$ small matrix blocks according to block-cyclic mapping. This approach ensures efficient parallel computation by storing the local matrix segments contiguously with a fixed leading dimension.

III. The Cloud Brain II AI Supercomputer

A. Ascend NPU: DaVinci Architecture

Unlike CPUs and GPUs designed for general-purpose computing or ASICs tailored for specific algorithms, the Ascend NPU represents a novel class of AI chips specialized for versatile neural network computations [9]. Central to the Ascend AI NPU's computing capability is the AI core, which is built upon the innovative DaVinci architecture. This architecture integrates matrix computing units (cube), vector computing units (vector), and scalar computing units (scalar) to enhance versatility and efficiency for AI computing. Furthermore, the Ascend NPU supports multiple precision calculations to accommodate varying data precision requirements, offering coverage for various computational demands in AI environments. Illustrated in Fig. 3, DaVinci architecture facilitates efficient matrix calculations, particularly in deep learning neural networks, with the Cube capable of executing 16×16 matrix multiplications with FP16 precision per operation (extendable to $16 \times 32 \times 16$ with INT8 precision inputs). Each Ascend 910 NPU is equipped with 32 AI cores.

The architectural design of the NPU AI core incorporates a multi-tiered memory hierarchy aimed at optimizing data access and processing efficiency. Of particular note is the role of the L1 buffer, serving as a sizable data cache within the AI core, strategically caching frequently accessed data to mitigate the frequency of read and write operations from the primary memory. Detailed specifications regarding the memory organization of the Ascend 910 model are given in Table I. Furthermore, the L0A and L0B Buffers are designated for the storage of inputs for cube computing unit instructions, while the L0C Buffer is allocated for the storage of resulting outputs from cube computing unit operations. Additionally, the unified buffer fulfills the role of a cache for vector and scalar operations. Collectively, the multi-level architecture of the Ascend NPU is meticulously optimized to accommodate the demands of high-performance neural network computations, ensuring efficient management of data movement and processing operations.

TABLE I
Multi-level memory hierarchy specifications for Ascend 910 NPU

Memory level	Size	Nominal bandwidth
Global memory (GM)	32 GB	GM → L1: 1.2 TB/s GM → UB: 1.2 TB/s
L1	32 MB	N/A
L0A/L0B	2 MB	N/A
L0C	8 MB	L0C → GM: 1.0 TB/s L0C → UB: N/A
Unified buffer (UB)	8 MB	UB → GM: 1.2 TB/s

TABLE II
Specification Comparison between NPU and GPU

	Huawei Ascend 910	Nvidia A100
Tensor unit dimension	$16 \times 16 \times 16$	$8 \times 4 \times 8$
On-chip cache	34 MB	40 MB
AI Flops	256 T	312 T

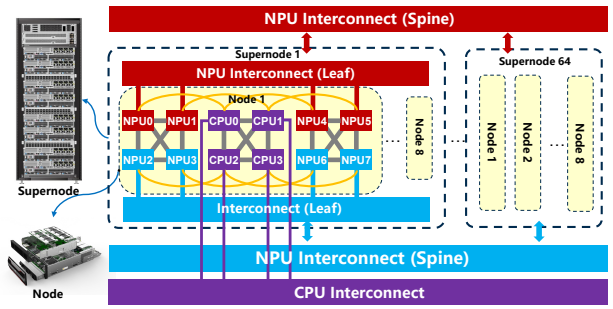

Fig. 4. AI computing system hierarchy for Cloud Brain II

Table II lists a comparison of specifications between the Huawei Ascend 910 and the Nvidia A100 GPUs. While the compute performance of the Huawei Ascend 910 slightly trails that of the Nvidia A100, its innovative large-scale tensor acceleration units and high-performance cache architecture makes it well-suited for extensive tensor computations.

B. AI Computing System Hierarchy and Cluster

Cloud Brain II addresses the complex issue of multi-chip collective communication through the implementation of an innovative interconnection strategy encompassing all AI chips within the system. Additionally, the architecture incorporates two distinct networking solutions: the CPU network and the NPU network, each tailored to accommodate the varied demands associated with CPU and GPU interconnections. as illustrated in Fig. 4. Each node of Cloud Brain II consists of 192 Kunpeng CPU cores (based on the Arm architecture) and 8 Ascend AI processors (NPUs).

Halving and doubling broadcast [29] is implemented in Cloud Brain II for inter-NPU communication across nodes. This distributed communication algorithm enables pairwise communication between NPUs, with each NPU exchanging data with its counterpart at each step until all NPUs complete the reduction operation. Notably, for systems with N NPUs, if N is a power of 2, only $log_2(N)$ communication steps are needed to complete the process. This algorithm effectively

mitigates single-node bottlenecks by optimizing bidirectional bandwidth utilization for both sending and receiving operations. Additionally, MPI serves as the communication library for inter-CPU communication across nodes.

IV. Methods

A. A Multi-Iteration Fusion Method for GEMM

The escalating computational prowess of AI chips has rapidly outpaced the corresponding growth in memory bandwidth, leading to the emergence of the "memory wall" challenge [16]. In the context of LU factorization, specifically during the trailing matrix update step depicted in Fig. 1, the long and narrow dimensions of input matrices L_{21} and U_{12}, alongside the substantial size of the trailing submatrix awaiting update, contribute to this challenge. Consequently, the frequent read and write operations to global memory during the iterative submatrix updates $A^{k+1} \leftarrow A^k - LU$ heavily constrain operational intensity, impeding overall processing speed. To mitigate these constraints, this work proposes a novel multi-iteration fusion method tailored to the GEMM operation. By fusing multiple iterative steps of GEMM into one, this method optimizes operational intensity and computational efficiency, harnessing the advantages of the low-latency, high-bandwidth on-chip memory. Furthermore, this method also overlaps GEMM computation with TRSM computation and the panel broadcast communication to effectively hide the communication cost.

To simplify the exposition, Fig. 5 displays a schematic illustrating a two-iteration fusion algorithm. In contrast to implementations on Summit and Frontier [27], where GPUs serve as the primary accelerators, our approach deviates by performing the diagonal update on the CPU, since the computation cost is relatively inexpensive. Furthermore, the TRSM operation is co-executed on both the CPU and the GPU, a methodology that will be elaborated upon in subsequent sections. We isolate the overlapping segment of the GEMM non-priority zones from the original two consecutive iterations as the GEMM non-priority zone. Conversely, the non-overlapping portion of the GEMM non-priority zones from these iterations forms the GEMM second priority zone, while retaining the GEMM priority zone, albeit renamed as the GEMM first priority zone for clarity. Consequently, the trailing submatrices are partitioned into three distinct regions: the GEMM first priority zone, the GEMM second priority zone, and the GEMM non-priority zone. Drawing inspiration from the look-ahead technique depicted in Fig. 2, we efficiently complete the diagonal update, TRSM, GEMM first priority update, and GEMM second priority update, with the last step being the GEMM non-priority update. It is worth noting that, for the sake of clarity, Fig. 5 depicts only the partitioning of the two-iteration fusion. However, the partitioning methodology remains applicable across multi-iteration fusion scenarios.

Drawing on the specific characteristics of the Ascend NPU's L1 buffer, which features a size of 1 MB per each of its 32 AI cores and supports FP16 matrix multiplication, the number of matrix blocks that can be accommodated simultaneously in the

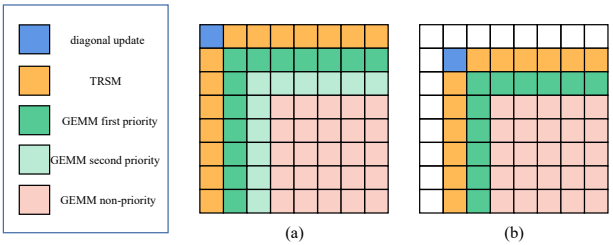

Fig. 5. Block LU factorization using two-iteration fusion (diagonal update on CPU, TRSM on CPU & NPU, GEMM on NPU)

L1 buffer depends on the block size. To illustrate without loss of generality, consider a 256×256 matrix block, where each block occupies 128 KB, allowing the buffer to accommodate up to 8 such blocks. For different block sizes, the number of matrix blocks that the L1 buffer can accommodate will vary accordingly, but the principle remains the same. In the L1 buffer, two buffer locations are allocated to implement the double buffering strategy [30] to facilitate efficient data transfer. Successively, $N_{\text{fused}} = 6$ (N_{fused} represents the number of iterations for fusion) upper panel matrix blocks are alternately transferred from global memory to L1. Upon completion of the current data block transfer and the initiation of computation, the next upper panel matrix block commences transfer simultaneously, enabling temporal overlap between data transfer and matrix computation. We apply the strategy of loading 6 lower panel matrix blocks and 1 upper panel matrix block during each computation. In the multi-iteration fusion computation described above, when computing the submatrix for the same row, once the necessary L matrices are moved into the buffer, subsequent updates to all matrix blocks in that row can reuse the cached L matrices without needing to repeatedly move them from the main memory. Over the course of 6 loading iterations, 6 upper panel blocks are successively loaded while reusing the 6 lower panel blocks, thereby facilitating the completion of computation for one block in the trailing matrix. The multi-iteration fusion operator, as depicted in Fig. 6, is implemented in this work. During the computation of a matrix block for the trailing matrix GEMM, a solitary read/write operation from the global memory suffices for the corresponding matrix block:

$$A^{k+N_{\text{fused}}} \leftarrow A^k_{\text{non-priority}} - \sum_{i=1}^{N_{\text{fused}}} L_i U_i$$

In contrast, the single-iteration approach necessitates six discrete global memory read/write operations.

The pipelines for the trailing matrix update on the NPU are delineated in Fig. 7. To summarize the computation process for a blocked matrix in a particular row in Fig. 7, the following steps are undertaken:

1) Initially, all required lower panel matrix blocks L_i for this row and for column j, where $i = [1, N_{\text{fused}}]$, are transferred into the AI core's L1 buffer. N_{fused} represents

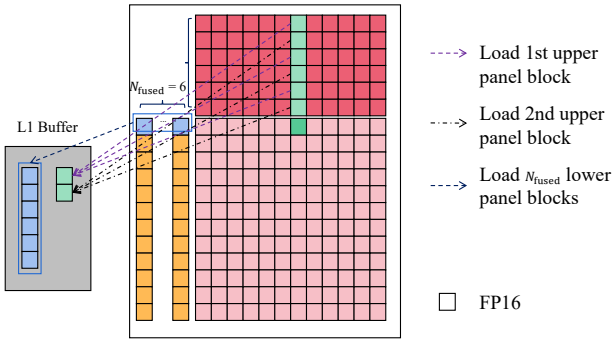

Fig. 6. Multi-iteration fusion for GEMM non-priority zone

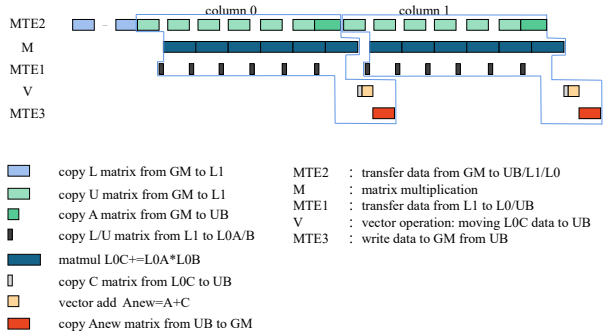

Fig. 7. Pipelines for GEMM non-priority zone

the number of iterations for fusion. $N_{\text{fused}} = 6$ when the block is size is 256×256.

2) Subsequently, the necessary upper panel matrix block U_i for this row is moved into the AI core's L1 buffer, where i is a single value from $[1, N_{\text{fused}}]$.
3) Next, one L_i matrix block and one U_i matrix block are transferred to the L0A/L0B buffer each time, situated closer to the AI core.
4) Following the previous step, matrix multiplication L_iU_i is executed on the AI core, concurrently with the memory copy of the new U_{i+1} matrix block.
5) As the process continues, loading of other U_i matrix blocks and matrix multiplication L_iU_i continue until completion. Concurrently, the matrix multiplication result C and the input matrix $A^k_{\text{non-priority}}$ are copied to the unified buffer.
6) Vector addition is performed on $A^k_{\text{non-priority}}$ and C.
7) Upon completion of the computation, the result $A^{k+N_{\text{fused}}}$ is written back to the Global memory and moves to column $j + 1$.

The advantages of fusing GEMM computations across multiple iterations within the GEMM non-priority zone are apparent, effectively reducing the number of global memory read/write operations associated with the trailing submatrices. This optimization strategy fully leverages the memory bandwidth of the L1 buffer, thereby significantly enhancing computational efficiency compared to the separate computation of multiple iterations. Furthermore, it is noteworthy that a larger L1 buffer facilitates the fusion of more iterations, thereby potentially expanding the scope of optimization.

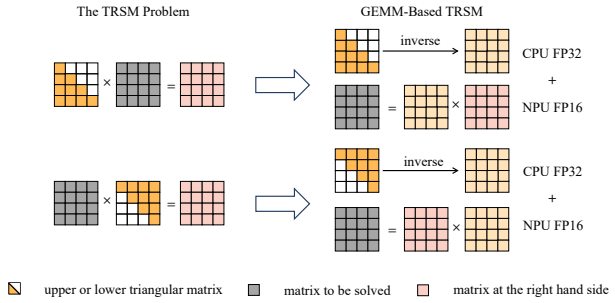

Fig. 8. Transformation of TRSM into GEMM-based TRSM

B. Co-Acceleration for TRSM using CPU/NPU

The triangular matrix equation is a mathematical problem widely encountered in various fields, characterized by its mathematical form as $op(A) \times X = \alpha \times B$ or $X \times op(A) = \alpha \times B$. Here, matrix A is an upper or lower triangular matrix, $op(A)$ denotes either A itself or its transpose, B is a given matrix, α is a constant, and X is the matrix to be solved for.

Existing methods for solving triangular matrix equations typically involve calling functions from libraries such as BLAS to solve them row by row, only utilizing the vector computing capability of AI chips, resulting in lower performance. In contrast, the proposed co-acceleration method for TRSM designs an algorithm that transforms the problem of solving triangular matrix equations row by row into a matrix multiplication problem, utilizing mixed-precision computing units including the CPU and the NPU to significantly improve the efficiency of solving triangular matrix equations, which can be seen in Fig. 8. In our approach, the small-scale matrix inversion within TRSM is executed on the CPU using FP32 precision, while the subsequent matrix multiplication is carried out on the NPU AI core using FP16 precision. This division of tasks capitalizes on the respective strengths of each processing unit. The rationale behind this transformation lies in the substantially superior matrix computation capabilities of AI chips compared to their vector computation capabilities.

If the triangular matrix solve's floating-point precision does not match the precision supported by the AI chip's matrix computing power, $\text{op}(A)^{-1}$ and B need to be converted to the supported precision and scaled numerically. For example, suppose $\text{op}(A)$ and B are in FP32, while the Ascend NPU AI core only supports FP16 matrix multiplication. The exponent lengths of different precisions are not consistent, with FP32 having a larger representable range than FP16, meaning that some values representable in FP32 cannot be accurately represented in FP16, requiring numerical scaling. The scaling process involves multiplying $\text{op}(A)^{-1}$ and B matrices by scaling factors s_1 and s_2, resulting in $s_1 \cdot \text{op}(A)^{-1}$ and $s_2 \cdot B$. The values of scaling factors s_1 and s_2 are determined based on the value ranges of $\text{op}(A)^{-1}$ and B, their floating-point precision, and the precision of the AI chip's matrix computing

power to avoid potential floating-point underflow and overflow after precision conversion.

After scaling, $s_1 \cdot \mathrm{op}(A)^{-1}$ and $s_2 \cdot B$ are converted to the lower precision supported by the AI chip, denoted as $(s_1 \cdot \mathrm{op}(A)^{-1})_{\text{low}}$ and $(s_2 \cdot B)_{\text{low}}$. The matrix multiplication result of these scaled matrices, $(s_1 \cdot \mathrm{op}(A)^{-1})_{\text{low}} \times (s_2 \cdot B)_{\text{low}}$, is computed using the AI chip's matrix computing unit. Finally, the result is converted back to the original problem's floating-point precision and scaled to obtain:

$$X = (s_1^{-1} \cdot s_2^{-1} \cdot \alpha) \times (s_1 \cdot \mathrm{op}(A)^{-1})_{\text{low}} \times (s_2 \cdot B)_{\text{low}}$$

It should be noted that the novelty of this co-acceleration approach is not related to how numerical scaling is conducted, as these operations are integral components of the overall co-acceleration approach. Further details on the numerical scaling process can be found in Ootomo et al. [31].

C. A Multi-Stage Multi-Level Heterogeneous Pipeline for LU Factorization

In large-scale computing systems employing GPUs as accelerators, LU factorization can be efficiently executed solely on GPUs. In this context, tensor cores can be leveraged for GEMM computations, while diagonal updates and TRSM operations can be performed using CUDA cores which are more versatile. Conversely, for specialized accelerators such as TPUs or NPUs tailored for neural network computations, their strength lies in low-precision matrix multiplication but may lack the versatility of CUDA cores. Therefore, when devising strategies to offload GEMM computations in LU decomposition to AI chips while delegating other tasks unsuitable for AI chips to CPUs, meticulous design of heterogeneous pipelines is crucial to ensure efficient collaboration among diverse computing units to complete the entire task.

Fig. 9 (a) depicts a basic multi-stage pipeline for LU factorization integrating the co-acceleration technique for TRSM between the CPU and NPU. In this setup, when the CPU is engaged in computations, the accelerator is forced to wait for the CPU's results, and conversely, leading to a scenario where both components alternate in waiting for each other. To optimize practical computations, leveraging asynchronous parallelism is crucial. With the CPU and NPU functioning independently, the pipeline can be optimized for concurrent task execution and resource utilization. This involves overlapping computation and communication tasks, enabling simultaneous CPU and NPU operation on independent segments. However, this is a simple pipeline but not be suitable for practical use.

Moving forward, we can devise a multi-stage pipeline inspired by the look-ahead technique, leveraging asynchronous parallelism to mitigate overheads, as illustrated in Fig. 9 (b). In this pipeline, the CPU host undertakes the tasks of executing diagonal updates and computing the matrix inverse for TRSM at iteration $i+1$, while the Ascend AI core handles matrix multiplication or GEMM at iteration i. This allocation of tasks allows the significant computational workload of GEMM on the NPU to effectively hide the computational overhead on the CPU. However, it is essential to note that this pipeline may encounter challenges, particularly if the memory overhead becomes a limiting factor due to bandwidth constraints.

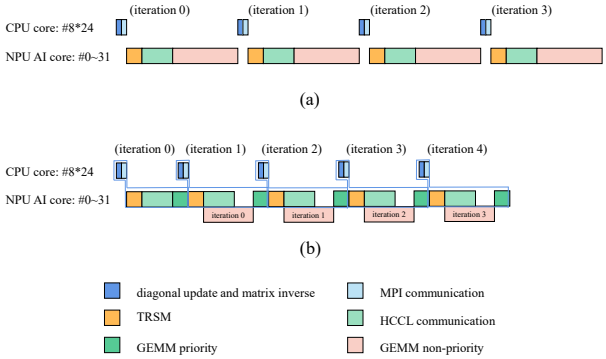

Fig. 9. Multi-stage pipeline for LU factorization

Progressing further, Fig. 10 depicts the multi-level, multi-stage heterogeneous pipeline for LU factorization, as proposed in this work. The parameters N_{fused} and n_{fusion} are key tunable variables, representing the number of fused iterations and the allocation of AI cores dedicated to the GEMM non-priority (fusion) region, respectively. This architectural design incorporates the multi-iteration fusion method introduced earlier. It is worth recalling that the GEMM computation is partitioned into three segments: GEMM first priority, GEMM second priority and GEMM non-priority. With the exception of the GEMM non-priority zone, the interdependencies and sequential nature of other computational or communication tasks remain consistent with the pipeline depicted in Fig. 9 (b). During iterations i to $i + N_{\text{fused}} - 1$, GEMM non-priority computation can be overlapped with diagonal updates on the CPU, MPI communication on the CPU, TRSM, HCCL communication, and GEMM priority computations on the NPU, which occur from iterations $i + N_{\text{fused}}$ to $i + 2N_{\text{fused}} - 1$. Given the comparatively higher computational cost associated with GEMM non-priority computation compared to TRSM and GEMM priority computation, it is reasonable to allocate a smaller proportion of AI cores to TRSM and GEMM priority computation while assigning a larger share to GEMM non-priority tasks. The pipeline architecture presented in this study effectively accommodates the interdependencies among various computational steps.

The coarse-level pipeline consists of different kernels or operators, including diagonal updates, matrix inversion, and MPI communication executed on the CPU, as well as TRSM and GEMM computations in the first priority and second priority zones on a few NPU AI cores, and GEMM computations in the non-priority zone on most AI cores, along with HCCL communication. The fine-level pipeline occurs within the GEMM computations in the non-priority zone, leveraging the memory hierarchy design of the NPU AI cores to efficiently utilize on-chip memory for GEMM operations. By designing such a multi-level (kernel level + intra-operator level), multi-stage pipeline on heterogeneous computing units

composed of the CPU and the NPU, the high performance of the NPU AI cores can be unlocked.

Compared to solutions using only CPUs or only AI chips, the multi-level, multi-stage heterogeneous pipeline for LU factorization proposed in this work offers broad applicability. The tunable parameters within the pipeline, such as N_{fused} and n_{fusion}, allow for adaptable optimization, enabling the AI chip, provided it supports low-precision matrix multiplication, to collaborate efficiently with the CPU in executing mixed-precision LU factorization.

D. HPL-MxP Computation Process

Fig. 11 illustrates the comprehensive HPL-MxP computation process involving both CPU and NPU. The heterogeneous CPU/NPU HPL-MxP algorithm, depicted in Algorithm 1, begins with LU decomposition of matrix A. In each iteration, the upper-left submatrix undergoes decomposition into L and U matrices on the CPU, followed by the computation of their inverses. These inverse matrices are then broadcast via MPI_Bcast to participating processes for subsequent computations. For TRSM, the processes transfer the corresponding inverse matrix to the NPU, facilitating panel updates, and broadcast the outcomes to the relevant process groups through HCCL. The utilization of the multi-iteration fusion method involves partitioning the trailing submatrices into distinct zones: the first and second priority zones, along with the non-priority zone (fusion zone). Initially, data within the first priority zone undergoes GEMM updates, followed by the immediate initiation of the next iteration of factorization. This process allows computations and communications to overlap with updates in other zones, a strategy known as the look-ahead technique. The second priority zone receives updates in every iteration, while the fusion regions are updated every N_{fused} iterations. Within the fusion zone's update phase, double-buffering mechanisms and matrix multiplication operations are leveraged to update the values of the trailing submatrix A. Upon completing LU decomposition, iterative methods on the CPU can be employed to solve the linear system of equations.

The transferability and adaptability of the proposed LU factorization methods to other architectures hinge on optimizations across multiple layers, as depicted in Fig. 12. At the application layer, the multi-iteration fusion technique significantly reduces global memory traffic, making it well-suited for architectures (TPU, FPGA, DSAs) featuring tensor units, high-speed on-chip memory, and deep memory hierarchies. This optimization, however, requires adaptation to micro-architectural factors such as tensor unit dimensions, memory hierarchy, memory bandwidth, and memory capacity across different hardware platforms. Specifically, the tensor unit dimensions dictate the optimal block size and tiling strategy necessary for achieving peak performance. The kernel-level heterogeneous pipeline involves the optimization of multiple computational kernels, communication tasks, and load-balancing mechanisms, rendering it adaptable to diverse architectures. Nonetheless, achieving load balance and maximizing pipeline efficiency necessitate careful allocation of compute resources tailored to the architecture's specific capabilities. Fine-tuning the intra-operator pipeline is essential for enhancing throughput by minimizing stalls and latency, but it is highly dependent on specific micro-architectural characteristics, such as pipeline depth, instruction-level parallelism, and requires targeted optimizations, including instruction-level tuning, memory prefetching strategies, and meticulous pipeline design. Although data distribution, layout, and access patterns are influenced by both the application and the underlying hardware, these aspects are secondary in terms of the contributions of this work. Intermediate system layer optimizations, such as communication libraries, task scheduling, kernel fusion, dynamic memory management, and graph optimization, can have a significant impact on overall performance and scalability across various computing environments.

Algorithm 1 CPU/NPU HPL-MxP Algorithm

for iter $= 0$ to $N - 1$ **do**
  (1) Diagonal Update
  Diagonal update using CPU FP32
  Matrix inverse using CPU FP32
  (2) TRSM
  Broadcast inversed matrix using MPI
  Transfer inversed matrix to the NPU
  Perform matrix multiplication in TRSM using FP16
  Intra-group broadcast using HCCL
  (3) Trailing Matrix Update
  Perform GEMM update for first priority using FP16
  Perform GEMM update for second priority using FP16
  Non-priority Update
  for $i =$ core_id to $n_{\text{row}} - 1$ with step n_{core} **do**
    N_{fused} L matrix blocks: GM $\rightarrow$ L1
    for $j = 0$ to $n_{\text{col}} - 1$ **do**
      for $i_t = 0$ to $N_{\text{fused}} - 1$ (double buffer) **do**
        The i_t-th U matrix block: GM $\rightarrow$ L1
        for $k = 0$ to $N_B/MatK$ (double buffer) **do**
          L and U matrices: L1 $\rightarrow$ L0A, L0B
          Matrix multiply using FP16 to obtain C
        end for
      end for
      $A_{\text{non-priority}}$: GM $\rightarrow$ UB
      C: L0C $\rightarrow$ UB
      Perform an add for $A_{\text{non-priority}}$ and C
      Added result: UB $\rightarrow$ GM
    end for
  end for
  LU factorization complete
end for
(4) Iterative Refinement
Initialize a good guess for x
while error $> err_{tol}$ **do**
  Compute residual $r = b - Ax$. Exit if $error <= err_{tol}$
  Solve for $LUx' = r$
end while

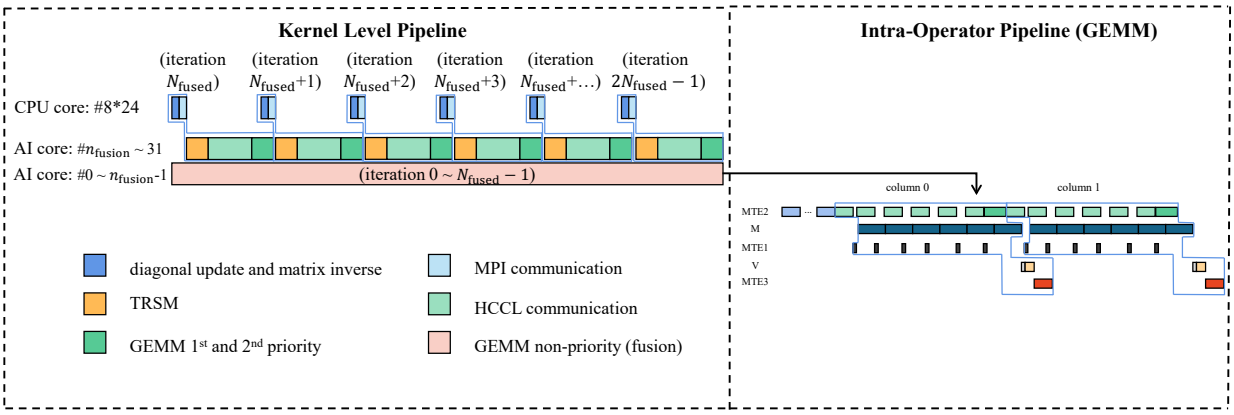

Fig. 10. Multi-level, multi-stage heterogeneous pipeline for LU factorization

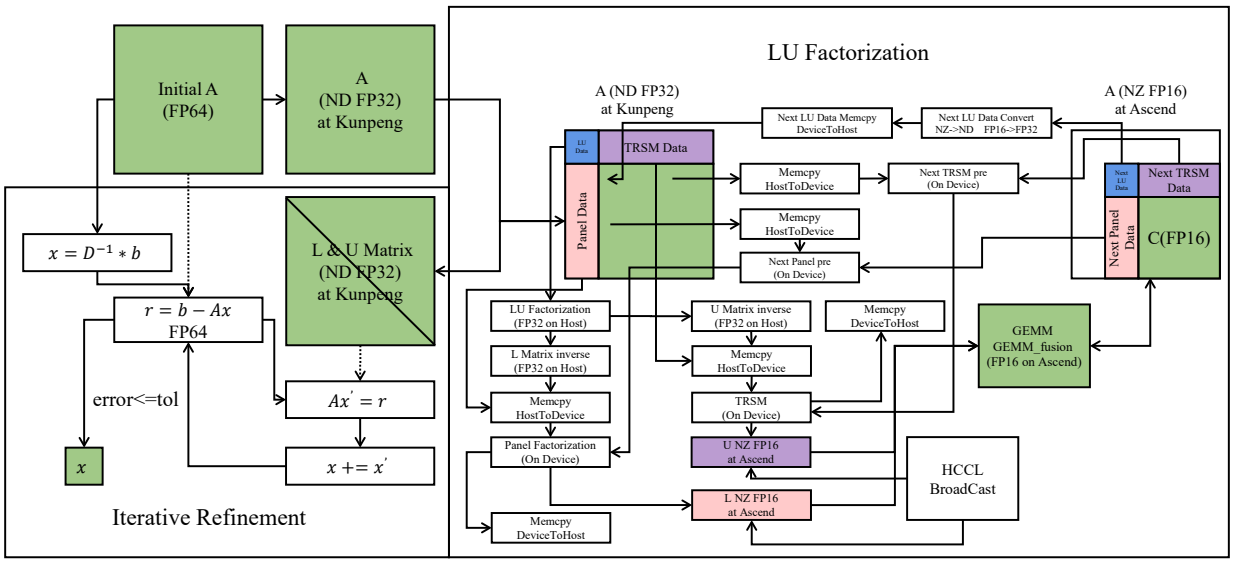

Fig. 11. Computation process on CPU and NPU

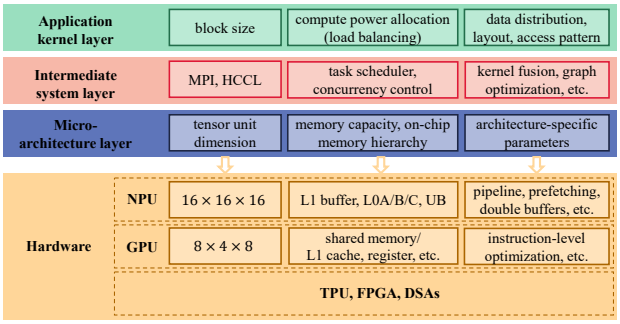

Fig. 12. A hierarchical view of transferability and adaptability of the proposed LU factorization methods to other architectures

V. Performance Results

To validate the effectiveness of the multi-iteration fusion method, HPL-MxP was implemented on Cloud Brain II, and the general matrix multiplication (GEMM) operation was optimized using the fusion technique across multiple iterations. Fig. 13 shows a roofline analysis for GEMM with varying block sizes on a single Ascend NPU, which theoretically peaks at 256 TFlops, following the methodology outlined by Williams et al. [32]. Conventional GEMM operations without fusion, utilizing block sizes of 256 × 256 (GEMM256) and 512 × 512 (GEMM512), achieved measured performances of 138.6 TFlops and 217.9 TFlops, respectively. The multi-iteration fused GEMM operator with a block size of 256 × 256 (GEMM256_fusion) significantly enhanced performance to 240.2 TFlops, slightly surpassing the GEMM512 operator and approaching the theoretical peak of the chip. Moreover, the operation intensity for the 256 × 256 block size dramatically increased from 100 Flop/Byte to over 600 Flop/Byte following the application of the multi-iteration fusion technique. These results underscore the substantial improvements in compute resource utilization, particularly the exploitation of on-chip memory, thereby maximizing the performance of AI chips.

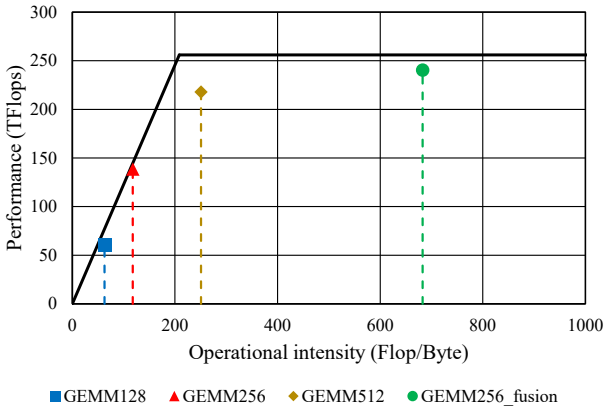

Fig. 13. Roofline for GEMM on Ascend NPU

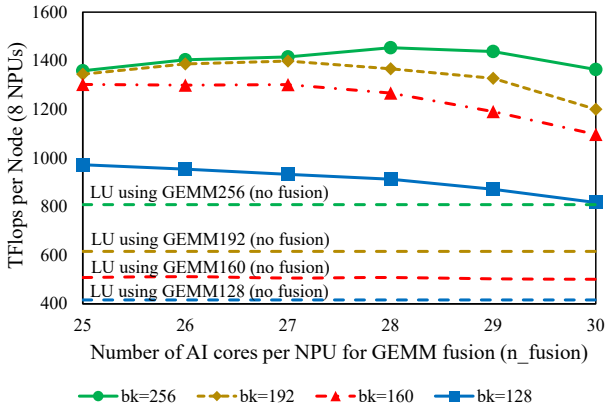

Fig. 14. Impact of block size and n_{fusion} on LU performance

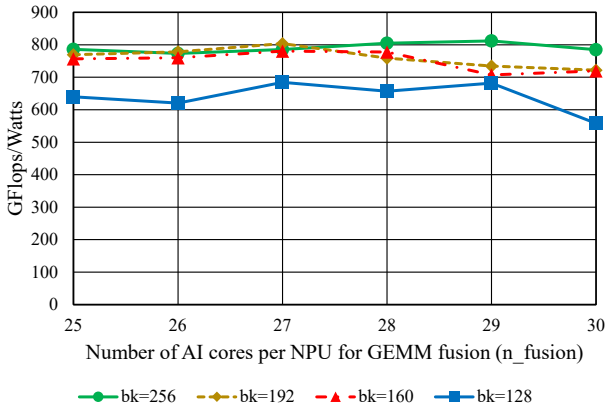

Fig. 15. Energy efficiency measurement of LU on one node (8 NPUs)

Fig. 14 shows the impact of block size (bk) and the allocation of AI cores dedicated to the GEMM non-priority/fusion region (n_{fusion}) on LU factorization performance on a single node. The horizontal axis represents the number of AI cores allocated to the GEMM non-priority/fusion region per NPU, while the vertical axis shows the measured performance of LU decomposition on the node. Various block sizes were tested in this study. The results indicate that larger block sizes result in better LU performance. This is because larger block sizes require fewer fusion steps, leading to more compact computation in both the fusion and priority regions. Consequently, there is less need for repeatedly launching kernels with small computational loads. This concentration improves the execution efficiency of kernels within the pipeline, resulting in enhanced overall performance. The optimal allocation of AI cores to the GEMM fusion region within the pipeline depends on the block size, with each size having a specific optimal value. For instance, when the block size is 256 and $n_{\text{fusion}} = 28$, the remaining 4 AI cores are allocated to TRSM and other computations. This allocation ensures balanced load distribution between the priority and non-priority regions, computation and communication overhead within the pipeline, minimizing pipeline stalls and maximizing overall performance. To balance the load in the pipeline, the number of AI cores allocated to the GEMM fusion region needs to be appropriately adjusted.

Fig. 15 shows the energy efficiency measurements for LU factorization on a single node comprising 8 NPUs. The power measurements, obtained using the `npu-smi` command, are specific to the NPU processors, excluding contributions from other subsystems. Across all test scenarios, energy efficiency ranged from 550 GFlops/Watt to 820 GFlops/Watt, demonstrating that the effective utilization of 32 AI cores, coupled with appropriate blocking strategies, can significantly enhance energy efficiency. Notably, the highest energy efficiency was observed at a block size of $bk = 256$, corresponding to improved AI core efficiency. Specifically, the energy efficiencies at $n_{\text{fusion}} = 28$ and $n_{\text{fusion}} = 29$ were measured at 805 GFlops/Watt and 812 GFlops/Watt, respectively. These results underscore the critical role of block size and AI core allocation in optimizing energy efficiency within NPU-accelerated computations.

Fig. 16 illustrates the ratio of measured performance to theoretical peak performance during the execution of HPL-MxP on a single node, with all curves corresponding to a block size of 256. The curve labeled `no fusion` represents the baseline performance without the application of the multi-iteration fusion optimization, exhibiting the longest runtime—approximately 1.8 times longer compared to when fusion is applied. This baseline performance is markedly inferior to that achieved with the multi-iteration fusion technique. The figure further demonstrates the impact of allocating different numbers of AI cores to the GEMM non-priority/fusion region on overall application performance. Notably, when $n_{\text{fusion}} = 28$ or $n_{\text{fusion}} = 29$, the overall performance is optimized, consistent with Fig. 14.

Fig. 17 illustrates the measured performance-to-theoretical performance ratio during the execution of HPL-MxP across multiple nodes of Cloud Brain II, ranging from 1 to 32 nodes. Each node of Cloud Brain II is equipped with 192 CPU cores and 8 Ascend AI accelerators (NPUs). The horizontal axis represents time, while the vertical axis represents the real-time measured performance-to-theoretical performance ratio.

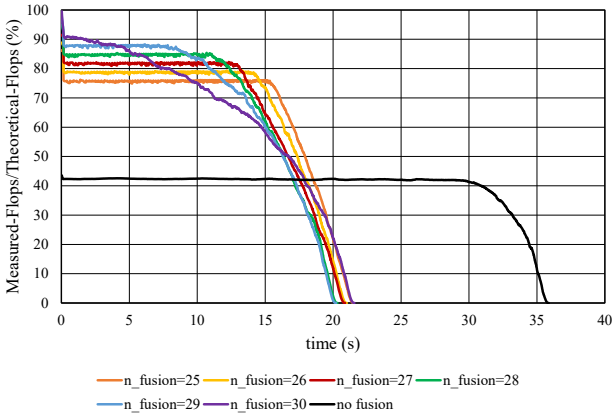

Fig. 16. Impact of n_{fusion} on HPL-MxP performance

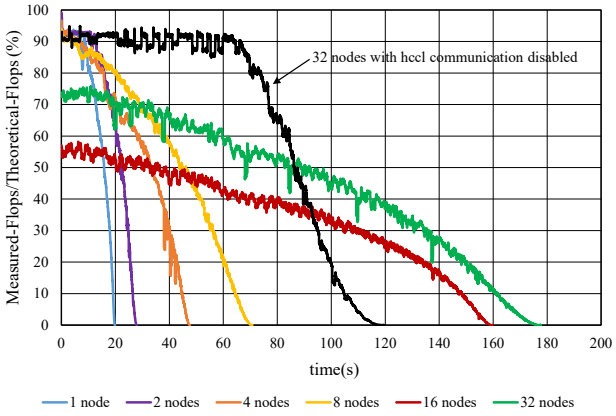

Fig. 17. Measured Performance of Multi-node HPL-MxP

It is observed that as the matrix size for LU factorization decreases, the measured utilization efficiency correspondingly declines over time. As shown in Fig. 17, in the initial phases of LU factorization, the performance-to-theoretical performance ratio for HPL-MxP exceeds 90% for configurations utilizing up to 8 nodes. Even at the scale of 32 nodes (256 NPUs), the measured-to-theoretical performance ratio remains above 70%. The observed performance degradation at 16 and 32 nodes may be attributed to increased inter-node HCCL communication overheads encountered during runtime. On Cloud Brain II, the unavailability of a complete supernode necessitated the random selection of nodes from different supernodes, thereby escalating communication costs. Additionally, the 2D cyclic decomposition strategy contributed to this performance decline. At lower node counts, only U-TRSM results require inter-node communication, while L-TRSM results involve intra-node communication. However, as node count increases, both directions necessitate inter-node communication, further amplifying communication overhead.

Fig. 18 illustrates the weak scalability in terms of TFlops and utilization on Clould Brain II with low-bit NPUs and CPUs. As the number of NPUs increases, the performance improves steadily, as evidenced by the increase from 780

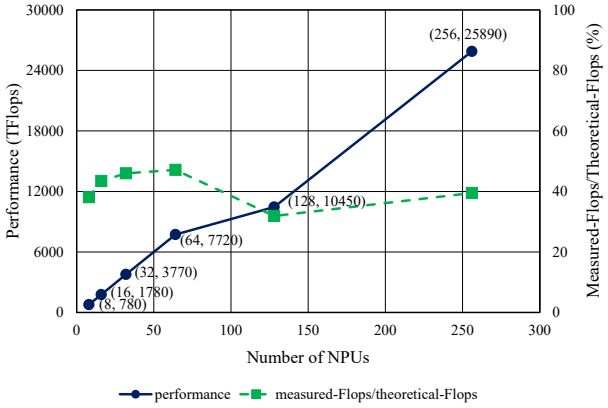

Fig. 18. Measured performance of Multi-NPU HPL-MxP

TFlops to 25890 TFlops. However, the utilization shows variations across different configurations. For instance, with 8 NPUs, the utilization is approximately 38.08%, which increases to around 47.12% with 64 NPUs. Interestingly, with 128 NPUs, the utilization decreases to approximately 31.89%, before slightly improving to about 37.52% with 256 NPUs. Overall, the scaling performance appears promising following the implementation of optimization techniques proposed in this study. However, in our experiments on the Cloud Brain II, constraints were encountered regarding the modification of BIOS settings, resulting in the CPU prefetch functionality remaining disabled. This circumstance notably impacted the performance of iteratie refinement on the CPU. Further testing on an isolated bare system with less NPUs showed that disabling CPU prefetch led to a reduction in the performance of iterative refinement by approximately 40%.

VI. Conclusion

In summary, the novel mixed-precision heterogeneous computing approach proposed in this work effectively integrates the high-precision computational capabilities of general-purpose CPUs with the high-performance, low-precision matrix multiplication capabilities of specialized AI chips. Through this innovative and synergistic approach, our method achieves mixed-precision LU factorization with significantly enhanced performance. Moreover, this research emphasizes the critical significance of in-depth comprehension and co-optimization of both application algorithms and AI accelerator architectures to attain optimal performance in today's complex computing landscape.

Acknowledgment

We would like to express our sincere gratitude to the reviewers for their valuable feedback and constructive comments, which significantly improved the quality of this work. This work is supported by the Pengcheng Laboratory Key Project (PCL2021A13) and the computing resources of Pengcheng Cloud Brain II. Weicheng Xue and Kai Yang contributed equally to this study, with Pengxiang Xu and Yonghong Tian as corresponding authors.

REFERENCES

[1] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 603–613.

[2] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, "Mixed-precision iterative refinement using tensor cores on gpus to accelerate solution of linear systems," *Proceedings of the Royal Society A*, vol. 476, no. 2243, p. 20200110, 2020.

[3] E. Carson and N. J. Higham, "A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems," *SIAM Journal on Scientific Computing*, vol. 39, no. 6, pp. A2834–A2856, 2017.

[4] N. J. Higham, S. Pranesh, and M. Zounon, "Squeezing a matrix into half precision, with an application to solving linear systems," *SIAM journal on scientific computing*, vol. 41, no. 4, pp. A2536–A2551, 2019.

[5] E. Carson and N. J. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, 2018.

[6] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, "Mixed precision block fused multiply-add: Error analysis and application to gpu tensor cores," *SIAM Journal on Scientific Computing*, vol. 42, no. 3, pp. C124–C141, 2020.

[7] J. Choquette, "Nvidia hopper h100 gpu: Scaling performance," *IEEE Micro*, 2023.

[8] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles *et al.*, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[9] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu, "Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 789–801.

[10] A. Boutros, S. Yazdanshenas, and V. Betz, "You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[11] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young *et al.*, "Scaling language models: Methods, analysis & insights from training gopher," *arXiv preprint arXiv:2112.11446*, 2021.

[12] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[13] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[14] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, 2004, p. 162.

[15] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2021, pp. 1–14.

[16] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *arXiv preprint arXiv:2403.14123*, 2024.

[17] M. A. H. Monil, N. R. Miniskar, K. Teranishi, J. S. Vetter, and P. Valero-Lara, "Matris: Multi-level math library abstraction for heterogeneity and performance portability using iris runtime," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1081–1092.

[18] G. Yuan, S. Palkar, D. Narayanan, and M. Zaharia, "Offload annotations: Bringing heterogeneous computing to existing libraries and workloads," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 293–306.

[19] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.

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

[21] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[22] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair, "Heterogeneous acceleration pipeline for recommendation system training," *arXiv preprint arXiv:2204.05436*, 2022.

[23] R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir *et al.*, "Optimizing cpu performance for recommendation systems at-scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[24] G. Tan, C. Shui, Y. Wang, X. Yu, and Y. Yan, "Optimizing the linpack algorithm for large-scale pcie-based cpu-gpu heterogeneous systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2367–2380, 2021.

[25] Q. Wang, L. Zheng, J. Zhao, X. Liao, H. Jin, and J. Xue, "A conflict-free scheduler for high-performance graph processing on multi-pipeline fpgas," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 2, pp. 1–26, 2020.

[26] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[27] H. Lu, M. Matheson, V. Oles, A. Ellis, W. Joubert, and F. Wang, "Climbing the summit and pushing the frontier of mixed precision benchmarks at extreme scale," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[28] S. Kudo, K. Nitadori, T. Ina, and T. Imamura, "Implementation and numerical techniques for one eflop/s hpl-ai benchmark on fugaku," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 2020, pp. 69–76.

[29] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[30] C. Yang, Y. Yao, Y. Lian, Y. Chen, R. Shah, X. Zhao, M. Chen, Y. Peng, and Z. Deng, "A double-buffering strategy to boost the lithium storage of botryoid mnox/c anodes," *Small*, vol. 15, no. 16, p. 1900015, 2019.

[31] H. Ootomo and R. Yokota, "Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance," *The International Journal of High Performance Computing Applications*, vol. 36, no. 4, pp. 475–491, 2022.

[32] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

Appendix: Artifact Description

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 A novel multi-iteration fusion method is proposed to mitigate the "memory wall" challenge encountered in GEMM computations. This method optimizes data transfer, emphasizing the utilization of the high-bandwidth L1 buffer on the NPU while minimizing global memory access.
- C_2 A co-acceleration strategy is devised to address the "mixed-precision optimization" challenge within TRSM computations. This method involves leveraging both the CPU and NPU resources, utilizing FP32 calculations on the CPU for less computationally intensive matrix inversions within TRSM, and FP16 operations on the NPU for matrix multiplication.
- C_3 A multi-stage, multi-level pipeline architecture is proposed for LU factorization, aiming to optimize performance in a heterogeneous computing environment. By strategically distributing computational tasks between the CPU and NPU, with GEMM operations executed on the NPU and other tasks on the CPU, this approach greatly enhances overall performance.

B. Computational Artifacts

As of August 2024, the HPL-MxP source code executed on Cloud Brain II is not publicly available due to existing regulations on Cloud Brain II. However, we intend to release it as open-source code in the future once any constraints on sharing the HPL-MxP source code or the customized image for HPL-MxP on Cloud Brain II are removed. As of August 2024, the only research item currently accessible for sharing is the HPL-MxP binary executable configured within a customized container image on Cloud Brain II. The HPL-MxP binary executable within the Cloud Brain II container environment, along with its guide for building, compiling, execution and analysis instructions, can be accessed at the following link:

A_1 https://doi.org/10.5281/zenodo.13316953

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 12-18 Figures 5-7
A_1	C_2	Figures 12-18 Figure 8
A_1	C_3	Figures 9-18

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

Artifact A_1 represents our implementation of HPL-MxP on Cloud Brain II. The code integrates several key optimizations: firstly, it employs the multi-iteration fusion method for GEMM, capitalizing on the high-bandwidth L1 buffer (C_1); secondly, it incorporates co-acceleration for TRSM utilizing both CPU and NPU, with the CPU handling less computationally intensive matrix inversions within TRSM and the NPU managing matrix multiplication (C_2); finally, it features a multi-stage, multi-level heterogeneous pipeline for LU factorization, encompassing kernel level and intra-operator pipelines as depicted in Figure 10 (C_3).

Expected Results

Performing experiments with the optimized HPL-MxP implmentation on Cloud Brain II is expected to yield valuable insights into its performance enhancements resulting from the implemented optimizations. Particularly, the comparison between experiments conducted with and without the multi-iteration fusion optimization for GEMM operations is expected to reveal notable differences in Floating Point Operations per Second (FLOPS), shedding light on the efficacy of this optimization technique. As depicted in Figure 13, the performance superiority of the fusion operator over its non-optimized counterpart is manifest, supporting conclusion C_1. Additionally, Figure 14 shows the impact of two tunable parameters of the multi-stage multi-level heterogeneous pipeline: block size and the number of AI cores assigned to the GEMM fusion zone, on the LU factorization performance using a single node comprising 8 NPUs. This performance evaluation shows that the pipeline method proposed in this work is tunable, supporting conclusions C_1 to C_3. Figure 15 shows power measurements for LU factorization on a single node with 8 NPUs, focusing exclusively on the NPU processors. Energy efficiency ranges from 550 to 820 GFlops/Watts, highlighting that optimal AI core utilization and appropriate block sizing, particularly with a block size of 256, significantly enhance efficiency, corroborating conclusions C_1 to C_3. The highest efficiency occurs when 28 or 29 AI cores are allocated to the fusion region, achieving 805 and 812 GFlops/Watts, respectively. Figure 16 shows the measured performance-to-theoretical performance ratio during the execution of HPL-MxP on a single node. It compares the overall HPL-MxP performance using multi-iteration fusion and no fusion, demonstrating that the best multi-iteration fusion method configuration outperforms the non-fusion method by more than 77%, supporting conclusions C_1 to C_3. Furthermore, deploying the highly optimized HPL-MxP code, which integrates the multi-iteration fusion method, CPU+NPU co-acceleration, and multi-stage, multi-level pipeline approach, across multiple nodes offers more

insights into its runtime behavior. Figure 17 provides a visualization of the dynamic ratio between measured FLOPS and theoretical FLOPS during execution, supporting conclusion C_1 to C_3. In addition, Figure 18 offers a comprehensive depiction of the ratio between measured FLOPS and theoretical FLOPS alongside hardware utilization efficiency across varying node configurations, further supporting conclusion C_1 to C_3.

Expected Reproduction Time (in Minutes)

- Artifact setup (customizing an image for HPL-MxP on Cloud Brain II): 200 minutes.
- Artifact execution: 150 minutes
- Artifact analysis: 650 minutes

Artifact Setup (incl. Inputs)

Hardware:

- Huawei Ascend 910A NPU (32 nodes × 8 NPUs/node)
- Huawei Kunpeng 920 CPU (192 cores/node, Arm architecture)
- PCIe 4.0 x16 (between CPU and NPU, 8 per node)
- 100 Gb RoCE v2 (1 per NPU)
- NVMe SSD (19.2 TB/node)
- DDR4.0 (2 TB/node)

Software:

- cmake v3.21.4
- Kunpeng GCC 9.3.1
- OpenBLAS v0.3.18
- ucx v1.11.2
- openmpi v5.0.0rc2
- CANN 6.3.RC2

Datasets / Inputs: N/A.

Installation and Deployment: To leverage Cloud Brain II's computing resources, users must initially create their own containerized images for job submission. Comprehensive instructions on customizing a Docker image for HPL-MxP are available in the "Setup" section of a guide via the Zenodo link: https://doi.org/10.5281/zenodo.13316953.

Artifact Execution

The execution paradigm of HPL-MxP encompasses two primary tasks: block LU factorization and iterative refinement. Notably, the iterative refinement phase relies upon the availability of the L and U matrices derived from the preceding block LU factorization process as preconditioners. In our work, emphasis is primarily directed towards optimizing the block LU factorization stage, owing to its predominant temporal overhead within the HPL-MxP computational workflow. The block LU factorization consists of three tasks: diagonal update, triangular solve with matrix right hand side, and trailing matrix update. These tasks collectively contribute to the computational complexity inherent in the block LU factorization process, thereby necessitating targeted optimization strategies to enhance overall efficiency and performance.

T_1 Perform LU decomposition on the top-left submatrix of the current iteration step, and compute their inverse matrices

T_2 Broadcast the required inverse matrix from T_1 to communicators participating in TRSM, and copy it to the corresponding NPUs

T_3 Perform matrix multiplication in TRSM using FPl16 on the NPU to obtain left and right panels, with the necessity of the inverse matrix from T_2 serving as input.

T_4 Broadcast left and right panels (from T_3) within communicators according to 2D cyclic decomposition

T_5 Perform GEMM update for first priority using FP16 (depends on T_4)

T_6 Perform GEMM update for second priority using FP16 for every iteration (depends on T_5)

T_7 Perform GEMM update for non-priority using FP16 on the NPU for every N_{fused} iterations (dependent on the results from T_1 to T_6 of the preceding N_{fused} iterations): 1) Move N_{fused} blocks on left panels from global memory to L1 buffer; 2) Move N_{fused} blocks on right panels from global memory to L1 buffer, one block at a time; 3) Move L and U blocks to L0A and L0B to perform matrix multiplication and move the result to the unified buffer; 4) Move trailing submatrix blocks to the unified buffer; 5) Perform addition operation between the matrix multiplication result and the trailing submatrix blocks; 6) Move the updated trailing submatrix blocks to global memory

T_8 Repeat the above T_1 to T_7 until the trailing submatrix is completely factorized, obtaining A=LU. Collect power data if needed

T_9 Calculate the initial value of $x = D^{-1} \times b$, where D is the diagonal matrix of A, and b is the right-hand side vector

T_{10} Utilizing MPI+OpenMP parallelism, compute the residual $r = b - Ax$ and reduces (the initial guess for x depends on T_9)

T_{11} Using the LU matrix obtained after the completion of T_8 as a preconditioner, solve LUx=r, and then update x, which necessitates the residual r computed in T_{10}.

T_{12} Calculate the error based on the reduced residual r. If the error is less than 16, stop and output

The experiments corresponding to each data point depicted in Figure 13 to Figure 18 are conducted repeatedly, with a minimum of three repetitions per data point. Subsequently, the results are averaged to mitigate potential variability.

Artifact Analysis (incl. Outputs)

The HPL-MxP implementation facilitates direct printing of measured FLOPS, as it records the total number of operations for the equation solve AX=b and the overall end-to-end execution time. Additionally, it calculates the operational intensity by tracking the total traffic byte from/to the global memory. Leveraging the multi-iteration fusion optimization proposed in this study for GEMM, FLOPs performance can approach the roofline, with the operational intensity improving from 100 Flop/Byte to over 600 Flop/Byte (block size = 256), which can be seen from Figure 13.

Larger block sizes and optimal AI core allocation significantly improve LU factorization performance on a single node with 8 NPUs, as shown in Figure 14. Larger blocks enhance the efficiency of GEMM fusion and priority tasks. Allocating 28 cores to GEMM fusion and the rest to TRSM tasks ensures a balanced workload and a highly efficient pipeline. Conversely, smaller block sizes necessitate more frequent kernel launches, leading to less optimal performance despite adjustments in AI core distribution.

Energy efficiency for LU factorization on a single node with 8 NPUs ranges from 550 to 820 GFlops/Watts, with the highest efficiency of 812 GFlops/Watts achieved using a block size of 256 and 29 AI cores assigned for fusion. These measurements, which were obtained with the npu-smi command for NPU processors only, are detailed in Figure 15.

Applying multi-iteration fusion during HPL-MxP execution on a single node yields a substantial performance improvement, cutting runtime by about 1.8 times compared to the baseline method without fusion, as shown in Figure 16. This enhancement is further refined by strategic allocation of AI cores to the GEMM fusion zone, with optimal performance achieved when 28 or 29 cores are allocated to this region.

The integration of multi-iteration fusion optimization, co-acceleration strategies for TRSM employing both CPU and NPU resources (192 CPU cores + 8 NPUs in each compute node), and a multi-stage, multi-level heterogeneous pipeline for LU factorization, as proposed in this work, enables the HPL-MxP code to unlock high performance leveraging low-bit NPUs and CPUs in unison. Evaluation across varying compute node configurations, as illustrated in Figure 17 and Figure 18, reveals insights into the measured-to-theoretical performance ratio, a metric indicative of hardware utilization efficiency. Specifically, during the initial phase of LU factorization, this ratio consistently surpasses 90% across up to 8 nodes on Cloud Brain II, each equipped with 8 NPUs, and maintains a commendable level exceeding 70% even at a higher node count of 32. Notably, the weak scaling efficiency exceeds 37.5% on a configuration utilizing 32 nodes, each equipped with 8 NPUs, totaling 256 NPUs. This assessment underscores the promising scalability and effectiveness of the optimization methodologies employed within our research endeavors.

For more in-depth exploration of artifact analysis, readers are encouraged to consult the "Analysis" section within the HPL-MxP guide on Cloud Brain II: https://doi.org/10.5281/zenodo.13316953.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Unlike traditional supercomputers, Cloud Brain II does not offer a terminal or pre-installed common high-performance computing software. Users are required to customize their own image, configure the environment locally within their image, and subsequently upload it to Cloud Brain II to submit their job.

- Register an account on Cloud Brain II
- Customize an image which includes installed software packages including Cmake, Kunpeng GCC, OpenBLAS, ucx, openmpi and CANN.
- Compile HPL-MxP in the image and push it to Cloud Brain II

For a detailed step-by-step guide, please refer to the "Setup" section at: https://doi.org/10.5281/zenodo.13316953.

Artifact Execution

To execute the HPL-MxP code on Cloud Brain II, users are required to complete the following steps:

- Log in to Cloud Brain II ModelArts to test debug mode functionality
- Create an execution script in OBS
- Submit a task in ModelArts
- run operator execution scripts

For additional details of the execution process on Cloud Brain II, readers are directed to the "Execution" section available at: https://doi.org/10.5281/zenodo.13316953.

Artifact Analysis (incl. Outputs)

To reproduce Figure 13, performance results for four different test cases of the GEMM operator (gemm128, gemm256, gemm512, and gemm256_6iter) are necessary. These results demonstrate a significant increase in operation intensity, from 60 to over 600 Flop/Byte, for the GEMM operation with a 256×256 block size after applying multi-iteration fusion. This optimization nearly reaches the roofline, surpasses the gemm512 operator, and enhances L1 buffer utilization despite memory wall challenges. Therefore, reproducing Figure 13 validates the contribution C_1.

To reproduce Figure 14, adjust the AI core allocation to the GEMM fusion zone and the block size. For each configuration, perform LU factorization on a single node with 8 NPUs, record the runtime, and compute TFlops by dividing the total operations by the runtime. For baseline configurations without fusion, adjust the block size, run LU factorization, and similarly calculate TFlops for comparison. This process confirms the contributions $C_1 \sim C_3$, as the LU factorization incorporates all the proposed performance optimizations, demonstrating that the optimized configuration consistently outperforms the baseline.

To reproduce Figure 15, run LU factorization on a single node with 8 NPUs, varying the block size. Use the npu-smi

command to measure the power consumption of the NPUs during execution. Compute energy efficiency in GFlops/Watt through dividing the performance in GFlops by the power consumption. Adjust the number of AI cores in the GEMM fusion region. This demonstrates how AI core utilization and block size impact energy efficiency, especially at larger block sizes.

To reproduce Figure 16, set the block size to 256 and run HPL-MxP on a single node, measuring the measured performance-to-theoretical performance ratio. The baseline "no_fusion" configuration, which does not apply multi-iteration fusion, will exhibit a runtime approximately 1.8 times longer than the optimized fusion method. Additionally, vary the number of AI cores allocated to the GEMM fusion zone to observe its impact on overall performance, with the best results typically occurring when 28 or 29 cores are allocated. This process validates the contributions $C_1 \sim C_3$.

To reproduce Figure 17, performance results using varying numbers of compute nodes are required, achieved by adjusting the node count in the HPL-MxP code. The time expense of each iteration step is recorded using std::chrono::high_resolution_clock and saved into a CSV file. Upon obtaining the time history for each compute node, Figure 17 can be plotted, utilizing the known theoretical floating-point operation count for each iteration step. Reproducing Figure 17 validates the contribution $C_1 \sim C_3$, since the HPL-MxP code on Clould Brain II applies all performance optimizations proposed in the manuscript.

To reproduce Figure 18, the total number of operations per second (OPS) for all test cases utilizing different numbers of compute nodes needs to be computed. Since we also have knowledge of the theoretical floating-point operations, it is straightforward to divide the measured flops by the theoretical flops to obtain the efficiency. Reproducing Figure 18 further validates the contributions C_1 to C_3.