Squeezing Operator Performance Potential for the Ascend Architecture

Yuhang Zhou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Zhibin Wang*
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Guyue Liu
Peking University
Beijing, China

Shipeng Li
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Xi Lin
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Zibo Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Yongzhong Wang
Huawei Technologies Co., Ltd.
Shenzhen, China

Fuchun Wei
Huawei Technologies Co., Ltd.
Shenzhen, China

Jingyi Zhang
Huawei Technologies Co., Ltd.
Shenzhen, China

Zhiheng Hu
Huawei Technologies Co., Ltd.
Shenzhen, China

Yanlin Liu
Huawei Technologies Co., Ltd.
Shenzhen, China

Chunsheng Li
Huawei Technologies Co., Ltd.
Shenzhen, China

Ziyang Zhang
Huawei Technologies Co., Ltd.
Shenzhen, China

Yaoyuan Wang
Huawei Technologies Co., Ltd.
Shenzhen, China

Bin Zhou
Shandong University
Jinan, China

Wanchun Dou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Guihai Chen
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Chen Tian*
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China

Abstract

With the rise of deep learning, many companies have developed domain-specific architectures (DSAs) optimized for AI workloads, with Ascend being a representative. To fully realize the operator performance on Ascend, effective analysis and optimization is urgently needed. Compared to GPU, Ascend requires users to manage operations manually, leading to complex performance issues that require precise analysis. However, existing roofline models face challenges of visualization complexity and inaccurate performance assessment. To address these needs, we introduce a component-based roofline model that abstracts components to capture operator performance, thereby effectively identifying bottleneck components. Furthermore, through practical operator optimization case studies, we illustrate a comprehensive process of optimization based on roofline analysis, summarizing common performance issues and optimization strategies. Finally, extensive end-to-end optimization experiments demonstrate significant model speed improvements, ranging from 1.07× to 2.15×, along with valuable insights from practice.

*corresponding authors

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716243

CCS Concepts: • **Computer systems organization** → **Processors and memory architectures**; • **Computing methodologies** → **Modeling methodologies**.

Keywords: AI Accelerator; Performance Modeling; Roofline Model; Operator Optimization

ACM Reference Format:
Yuhang Zhou, Zhibin Wang, Guyue Liu, Shipeng Li, Xi Lin, Zibo Wang, Yongzhong Wang, Fuchun Wei, Jingyi Zhang, Zhiheng Hu,

Yanlin Liu, Chunsheng Li, Ziyang Zhang, Yaoyuan Wang, Bin Zhou, Wanchun Dou, Guihai Chen, and Chen Tian. 2025. Squeezing Operator Performance Potential for the Ascend Architecture. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716243

1 Introduction

With the rapid evolution of deep learning technologies, numerous enterprises have devoted themselves to developing domain-specific architectures (DSAs) optimized for computations [22, 41], such as Google's Tensor Processing Units (TPU) [15, 20, 36] and Huawei's Ascend chips [23, 24]. As a representative, Ascend supports a wide range of influential models, such as DeepSeek V3/R1 [17, 25], and delivers outstanding performance and energy efficiency [40, 49, 50].

Compared to general-purpose GPU architectures, Ascend introduces several targeted architectural features to improve the performance of AI workloads:

- *Dedicated compute units.* Ascend assigns distinct roles to its compute units, with the Scalar unit handling control logic, and high-performance units like Vector and Cube dedicated to vector and matrix operations, respectively, enabling parallelism for different calculations.
- *Customized memory buffers and flexible data transfer.* Ascend is designed for AI-specific memory access and features customized memory buffers, including L0 A/B/C buffers for Cube and the unifier buffer for Vector. Data transfer is flexible, allowing for cross-layer and asymmetric bandwidth transfers rather than strict layer-by-layer flow. This setup lets users choose the transfer paths that best match workloads, minimizing overhead.
- *Efficient transfer control and instruction pipelining.* To prevent memory access conflicts, Ascend uses the memory transfer engine (MTE) that enforces serial execution within each MTE for transfers, while enabling parallel transfers between MTEs. Additionally, through instruction pipelining, Ascend supports broader parallelism, encompassing compute and transfer components. Within the same component, operations are executed sequentially, while they run in parallel across components. This design allows users to control pipelining explicitly for maximum efficiency.

While these designs grant users greater flexibility and performance potential, they also pose significant challenges in optimizing operator performance. Given that users must manually manage computation, transfers, and instruction pipelines, developers unfamiliar with Ascend's architecture can easily encounter poor performance. The causes are complex, including but not limited to: suboptimal algorithms, parameter configurations, or task allocations leading to inefficient computation, poorly selected transfer paths or bandwidths reducing MTE efficiency, and improper instruction sequence or too much/few synchronization impacting pipeline parallelism. Therefore, effective optimization first requires accurate performance analysis to pinpoint bottlenecks.

Unfortunately, existing performance analysis methods cannot effectively identify bottlenecks in Ascend operators. The mainstream Roofline models, such as DRAM Roofline [47, 53] and hierarchical Roofline [19, 38, 54] for CPUs and GPUs, are not well-suited for Ascend due to the significant architectural differences. Even naively extending the Roofline model from GPU to Ascend is not suitable for two reasons.

- *Numerous combinations of precision and transfer.* Ascend incorporates three compute units, each supporting multi-precision computation, totaling 9 precision-compute units. Moreover, it employs various memory buffers, interconnected by up to 20 memory transfers. A naive comparison of each precision to each transfer, analogous to GPU analysis, would yield 180 roofline combinations, complicating visualization and analysis.
- *Contention, asymmetry, and cross-layer interactions.* Ascend suffers contentions from the precision computations and memory transfers. For example, the MTE unit sequentially schedules transfers within it, while considering the transfers separately may misdiagnose underutilization when fully occupied. Moreover, the cross-layer transfer and asymmetric bandwidth complicate the analysis. Similarly, challenges also exist in mixed-precision computation within the same compute unit.

In this paper, we make the following contributions:
Bottleneck Analysis (Section 4). We introduce the concept of a "*component*" to describe the serial execution of instructions within it and the parallel execution between different components. Physically, each component corresponds to a specific compute or MTE unit. Utilizing this abstraction, we introduce a component-based roofline model to present whether the operator is bound by a component or underutilized. For underutilization cases, using the component execution time, we decomposed utilization into component active time ratio and execution efficiency and attributed underutilization to two primary causes. 1) *Insufficient Parallelism*: The low parallelism between components results in near-serial execution, leading to inadequate execution time and consequently low utilization. 2) *Inefficient Component*: Even if there is sufficient parallelism between components, low execution efficiency also results in low utilization.
Optimization Experience (Section 5). To facilitate operator optimization, we provide three in-depth case studies in our practice. These cases illustrate how we use Roofline analysis to identify bottleneck components, followed by inspecting pipeline status and operator code to diagnose root

causes, thereby enabling the application of targeted optimization strategies. The cases cover cube, vector, and scalar computations, as well as all MTE units, addressing multiple types of bottlenecks such as insufficient parallelism, inefficient compute/MTE, and MTE bound. Additionally, we summarize common operator bottlenecks and the corresponding optimizations that best address them.

Extensive Evaluation (Section 6). Through two years of practice, we have optimized 11 models from real-world deep learning training and inference tasks, including those in vision, NLP, recommendation systems, and large language models (LLMs), especially PanGu-α with over 100 billion parameters [56]. Performance optimization results show computation time speedups ranging from 1.08× to 2.70×, with overall acceleration between 1.07× to 2.15×. A total of 41 optimized operators have been integrated into the Ascend operator library. We present end-to-end analysis and optimization and share insights from extensive experiments on bottleneck distribution and optimization selection.

Scope: This paper focuses on the computational performance of operators, making I/O and communication bottlenecks in large-scale training orthogonal to this work.

2 Background and Motivation

In this section, we will first introduce the Ascend architecture and its unique design principles. However, such a distinctive design poses significant challenges for optimizing operator performance. Even worse, existing roofline analysis, including its naive extension, fails to pinpoint operator bottlenecks, nor can it provide guidance for optimization.

2.1 Ascend Architecture

As the mainstream accelerator, GPU provides considerable inspiration for Ascend's design, particularly its efficient heterogeneous compute units and hierarchical memory structure. Specifically, a GPU consists of multiple Streaming Multiprocessors (SMs), each equipped with numerous computing units, including CUDA Cores for general-purpose math operations and Tensor Cores for tensor multiplication-accumulation tasks. The hierarchical memory in GPU encompasses global memory, L2 cache, and L1 cache (shared memory). Data flow from global memory through the L2 cache into the L1 cache before being fed into computing units.

Unlike GPUs for general-purpose computing, deep learning DSA like Ascend is tailored from the ground up for DL, incorporating unique designs optimized for AI workloads.

Dedicated High-Performance Compute Units. In deep learning models such as CNN and Transformer, matrix operations (e.g., matrix multiply, convolution, and fully connected layers) account for 70-90% [45]. Regular compute units like CUDA Core are well-suited for basic operations like addition but lack the efficiency for these high-throughput tasks. Accordingly, high-performance units such as the tensor cores and the systolic array [18] can accelerate the computations. Similarly, Ascend adopts a heterogeneous design but with a more specialized focus: regular units handle logic, while dedicated high-performance units are tailored for vector and matrix operations. This highly specialized design aims to enable these units to work in parallel, boosting computation.

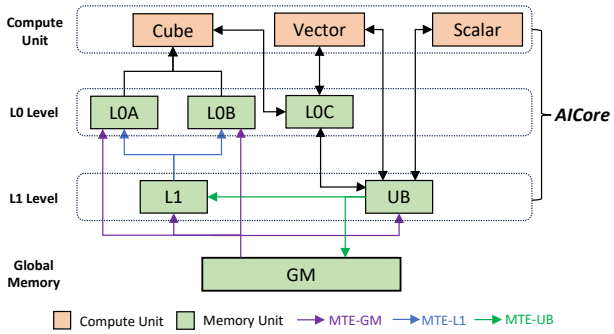


Figure 1. The architecture of AICore in Ascend.

Specifically, Ascend's computational core, AICore, integrates three types of compute units with varying precision[1], each specialized for a particular computing task: i) *Scalar Unit* (supporting INT32/FP16/FP32/FP64) similarly to a CPU Core, is primarily responsible for instruction control and logic operations; ii) *Vector Unit* (supporting INT32/FP16/FP32) follows the SIMD (single instruction multiple data) and executes most vector-related computations, such as normalization, softmax, pooling and activation. Finally, iii) *Cube Unit* (supporting INT8/FP16) is dedicated to accelerating compute-intensive operations, i.e., matrix multiplication (MatMul).

Optimized On-Chip Memory and Data Transfer. Although dedicated compute unit designs offer higher computational limits, achieving optimal performance requires fast data provisioning. As shown in Figure 1, Ascend also adopts three levels of the memory hierarchy, from the top to the bottom: L0, L1, and Global Memory (GM)[2]. However, unlike the GPU, Ascend incorporates extensive memory buffers and flexible data transfer options to minimize access overhead.

- *Customized memory buffers:* Ascend provides multiple customized buffers for specific compute units, enabling efficient data inflow and outflow. Specifically, the L1 level distinguishes between Cube and Vector/Scalar computations through L1 Buffer and Unified Buffer (UB). First, to accelerate matrix multiplication, the memory involved in Cube computations is divided into L0A, L0B, and L0C Buffer for two inputs and one output, respectively. For example, in a fully connected layer, the two input matrices (feature map and weights) are transferred from the GM to the L0A and L0B Buffer. The Cube unit multiplies these matrices and writes the output to the L0C

[1]These precision standards come from the Ascend training chip, and the overhead of type-casting is accounted for the compute or MTE units.
[2]We omit registers and external memory, e.g., CPU memory and SSD.

Buffer, which then feeds into the Vector unit for activation (e.g., ReLU), minimizing unnecessary data transfers. Meanwhile, the UB, as a flexible shared memory among different compute units, temporarily holds various data, including feature maps, weights, and activations, ensuring data is available for subsequent operations.

- *Cross-layer transfers and asymmetric bandwidth:* Ascend also enhances data transfer options to better align with AI workload characteristics. Unlike GPUs, which only allow transfers between adjacent memory layers, Ascend enables cross-layer transfers. For example, in addition to the typical data flow from GM to L1 ($GM \rightarrow L1$), data can also bypass L1 and be transferred directly to L0A/B Buffer to speed up Cube computations. Moreover, recognizing the substantial size difference between two inputs in matrix multiplications, such as the large feature map stored in L0A and the relatively smaller weights in L0B, Ascend offers higher bandwidth from L1 to L0A than from L1 to L0B. This design enables prioritized bandwidth allocation based on operator requirements.

In a nutshell, the primary difference in memory access between Ascend and GPUs is that, while GPUs rely on auto data management via caches, Ascend's buffers provide greater flexibility. Developers can control what data is loaded into the buffer, when it is loaded, and to which compute unit it is transferred, allowing for higher efficiency.

Efficient Data Transfer Control and Pipelining. While diverse transfers enhance computation efficiency, we also note that there is contention between different transfers. For example, GM transfers but with different destinations ($GM \rightarrow \{L1, L0A/B, UB\}$) share the bandwidth of the GM. Meanwhile, the hardware enforces that only one outbound transfer can occur at any given time per unit. Therefore, Ascend introduces a physical control unit, *Memory Transfer Engine*, to schedule data transfers to avoid conflicts while maximizing parallelism. Specifically, transfers within the same MTE unit must execute serially, while transfers across different MTE units can operate in parallel. Figure 1 shows three MTE units in Ascend, each corresponding to transfers originating from GM, L1, and UB, respectively. The blue MTE-L1 controls the $L1 \rightarrow \{L0A/B\}$ transfers, and the purple MTE-GM controls $GM \rightarrow \{L1, L0A/B, UB\}$ transfers and the green MTE-UB controls $UB \rightarrow \{OUT, L1\}$ transfers.

Furthermore, Ascend provides an instruction pipelining mechanism on hardware to enable efficient execution between compute and MTE units. Specifically, both computation and transfers are implemented by corresponding instructions, which are dispatched to the Cube, Vector, Scalar, and three MTE queues based on the type, awaiting execution by the respective hardware units. Instructions within the same queue are executed sequentially, while those in different queues can be executed in parallel, forming the instruction pipeline. This pipeline is explicitly managed, allowing developers to optimize efficiency by adjusting instruction sequences and reducing synchronizations. However, incorrect execution order and dependency handling can result in excessive waiting times, which degrade operator performance.

2.2 Operator Optimization on Ascend Is Non-trivial

Ascend's unique architectural design offers significant power and flexibility, but makes optimization more complex. Consider a convolution operator, which involves matrix multiplications and additions handled by the Cube unit, the feature maps and weights should be transferred from GM to L0A and L0B for computation. Poor performance in convolution operators can be caused by the following issues.

The first issue is the inefficiency of the compute units, particular suboptimal algorithms and parameter configurations hinder their processing. Additionally, developers must assess whether task allocation across different compute units (Cube, Vector, Scalar) is appropriate. Each unit has its strengths, and incorrect assignments can reduce efficiency.

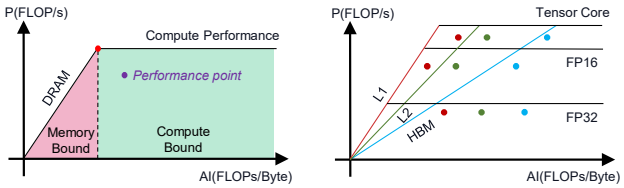
Another common cause is improper memory transfer selection. For example, larger feature maps can benefit from direct transfer to L0A, while weights could be staged in L1. Failing to make these data-specific choices can lead to unnecessary transfer costs. Furthermore, data allocation between L0A and L0B with different bandwidths can also go wrong. If developers place larger feature maps in the slower L0B and weights in the faster L0A, the Cube unit will wait longer.

Operator performance is also closely tied to inter-component parallelism, especially in Ascend, where the parallelism between computation and data transfer relies heavily on manual control by developers, making it prone to errors. For example, while computing one part of a matrix, the next part's data should be loaded from GM to L1 simultaneously. This ensures that the Cube unit can immediately receive the next input once they finish with the current data. However, if the $GM \rightarrow L1$ instruction is not correctly sequenced, parallelism will be disrupted. Additionally, developers must manually insert synchronization instructions, like *pipe_barrier*, to coordinate parallelism across components. Too much synchronization reduces the parallel efficiency between compute and transfers, while too few can result in data or cache conflicts.

The various potential issues with Ascend operators indicate that maximizing operator performance requires meticulous tuning across all aspects of both hardware and code. However, accurately identifying them through performance analysis becomes a challenging, yet essential task.

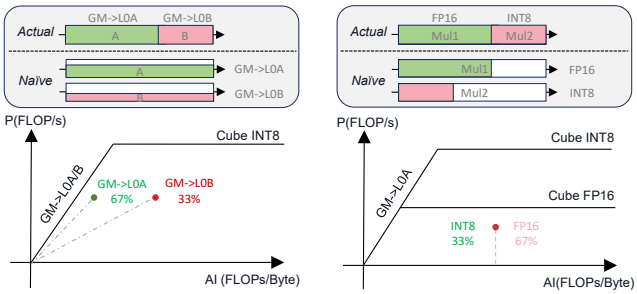
2.3 Limitations of Performance Analysis

The operator performance analysis is paramount for optimization, and previous studies [7, 11, 46, 55] have successfully done this on CPUs and GPUs. The most representative work is the roofline model [47], which is widely applied in performance analysis such as Empirical Roofline Toolkit (ERT) [53], Intel Advisor [19] and Nsight Compute [38].



(a) DRAM Roofline. **(b)** Hierarchical Roofline.

Figure 2. Existing roofline models.



(a) Bandwidth underutilization. **(b)** Compute underutilization.

Figure 3. Incorrect analysis cases of the naive roofline.

DRAM Roofline [47]. This model was originally conceived to evaluate the performance of CPU kernels accessing Dynamic Random Access Memory (DRAM). It assesses kernel performance by considering Floating-point Operations Per Second (FLOPs) and data accessed from DRAM (Byte). As depicted in Figure 2a, the *performance point* indicates the performance of this kernel. The X-axis represents Arithmetic Intensity (AI), which is the ratio of FLOPs to Byte. The Y-axis denotes Peak Floating-Point Performance (P), with a horizontal line (*arithmetic ceiling*) indicating the peak throughput. The slope of the diagonal line (*bandwidth ceiling*) reflects the Peak DRAM bandwidth. The ridge point i.e., the intersection of the horizontal and diagonal lines, as well as the dashed vertical line starting from it, divides the space into two regions. The left region is *memory-bound* as the performance is limited by memory bandwidth, while the right one is *compute-bound* as the performance is limited by arithmetic.

Hierarchical Roofline [54]. In practice, the CPU and GPU utilize a hierarchical architecture with multi-layer caches. As any layer of memory can be a bottleneck, the hierarchical roofline model is proposed to model them. To assess whether an operator's performance hits its bandwidth limitation at a specific layer, researchers construct individual bandwidth ceilings for each layer as shown in Figure 2b. Furthermore, the hierarchical roofline supports evaluating the arithmetic ceilings of various precision and compute units within a GPU, including FP32, FP16, and Tensor Core [13] operations.

Naive Roofline for Ascend. Due to the significant architectural differences between Ascend and CPUs/GPUs, original roofline models are inherently unsuitable for Ascend. To analyze Ascend performance, a naive method is to build rooflines by comparing each precision-compute unit against each transfer *independently*, similar to how the hierarchical Roofline expands from DRAM Roofline. For example, the roofline between the transfer of $L1 \rightarrow L0A$ and the FP16-Cube is built to identify whether the cache of the left matrix or the FP16 computation in Cube is the bottleneck in MatMul. However, this model overlooks the features of Ascend, which results in incorrect and complicated analysis.

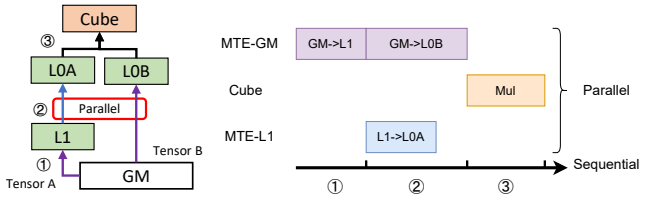
Issue 1: Complicated visualization and analysis due to massive combinations between percisions and transfers. Generally, according to Figure 1, the Ascend chip has 9 precision-compute units and 20 transfers, which results in 180 combinations for rooflines. The massive combinations result in crowded performance points when visualizing the roofline model, making it difficult to analyze.

Issue 2: Incorrect analysis due to the contention of memory transfers within the same MTE unit. Consider the example in Figure 3a, where two matrices, A and B, are multiplied, with A being twice the size of B. The matrices are transferred sequentially via $GM \rightarrow L0A$ and $GM \rightarrow L0B$ due to memory contention within the MTE-GM. Assuming that the MTE-GM remains fully occupied, both transfer bandwidths are fully utilized and equal. However, the naive roofline model assumes that data transfers of A and B are executed in parallel throughout the entire operation, overlooking the sequential nature. Since $GM \rightarrow L0A$ takes 67% of the time and $GM \rightarrow L0B$ takes 33%, the naive analysis mistakenly concludes that the bandwidth of both transfers is underutilized (67% and 33%, respectively).

Issue 3: Incorrect analysis due to the sequential execution of mixed-precision operations within the same compute unit. Similarly, a single compute unit may involve operations of multiple precisions executed sequentially. However, the naive roofline model only separates rooflines for different precisions, overlooking the serial nature. For example, as shown in Figure 3b, during data quantization, the Cube involves multiplication with FP16 and INT8 precisions, where the peak performance of INT8 is twice that of FP16. Assume the Cube is always busy and achieves the peak performance of given precision. When the two precision operations have the same number of operands, FP16 takes twice as long as INT8. However, the Naive analysis concludes that both achieve the same computational performance over the entire period (the same performance point) and their utilization is only 67% and 33%.

Re-visiting of the error analysis is detailed in Section 4.2.

In summary, the naive roofline not only loses the easy-to-visualize feature but also fails to distinguish between bottleneck and underutilization. As a result, developers urgently need an accurate solution for analyzing Ascend operator performance to effectively guide optimization efforts.



(a) Data flow. **(b)** Execution timeline.

Figure 4. The execution of matrix multiplication $A \times B$.

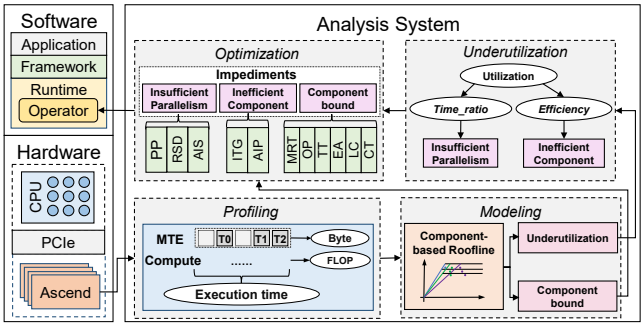


Figure 5. The workflow of analysis system.

3 Overview

In this section, we present two key observations that motivate our work and the workflow of our optimization system.

3.1 Key Observation

Abstraction of Component. To better understand the Ascend architecture, we demonstrate the data access flow of the matrix multiplication $A \times B$ in Figure 4a. In step ①, tensor A follows the $GM \rightarrow L1$ transfer. Subsequently, in step ②, tensor A executes the $L1 \rightarrow L0A$ transfer, while tensor B concurrently follows the $GM \rightarrow L0B$ transfer, achieving parallelism. In step ③, tensors A and B are fed into the Cube to compute $A \times B$. Correspondingly, Figure 4b depicts the execution timeline of MTEs and Cube.

In summary, operations with different precisions (or transfers) among various compute and MTE units ("*component*") can be parallelized, whereas those within the same component are sequential. This is attributed to the fact that *each component in Ascend corresponds to a physically existing instruction queue*, which sequentially schedules instructions within it, and different queues operate concurrently.

Additionally Profiling Metrics. The instruction queue for each component provides more detailed profiling metrics for analysis. The additional metrics come from two aspects:

- **Analyzing the instructions of each component: transfer bytes/operations of precision.** The instruction queue of each component can provide the number of each type of instruction, which can be further utilized to derive the number of bytes (operations) for each transfer (precision). These metrics reflect the importance of each transfer (precision) within the component, which can be further utilized to compute the ideal performance of the entire component and identify its bottleneck.
- **Monitoring the queue status: execution time of the component.** By monitoring the non-empty time of the instruction queue, we can estimate the actual execution (active) time of the component, which can be utilized to identify the cause of component underutilization, such as the low execution time or inefficient execution.

3.2 Workflow

Typically, the workflow for the analysis system, as depicted in Figure 5, involves the following concise steps:

Profiling. With the assistance of profiling tools provided by Ascend, such as msprof [3] and Pytorch Profiler [39], we have filtered the required metrics, including the transfer bytes/operations of each transfer (precision), the execution time of each component and operator time, for further analysis.

Modeling. (Section 4.1) Based on the abstraction of components, we propose the component-based roofline model to analyze the operator performance. This allows us to identify bottleneck components (component bound) or determine if all components are underutilized (underutilization).

Underutilization Analysis. (Section 4.2) For the underutilization cases, we further leverage the additional profiling metrics "execution time of each component", thereby decomposing the utilization into component active time ratio and component execution efficiency. This decomposition further pinpoints whether the underutilization is due to insufficient parallelism or inefficient component execution.

Optimization. (Section 5) After identifying the bottleneck causes, optimization becomes more targeted. We present three in-depth case studies of operator optimizations (Add_ReLU, Depthwise, and AvgPool) and share insights from practices.

4 Bottleneck Analysis

In this section, we will introduce an enhanced Roofline model that utilizes component abstraction for effective bottleneck classification and visualization. Further, based on the visual representation of component utilization, we conduct a detailed analysis of the root causes of underutilization.

Due to the nature of sequential execution within the same component and parallel execution between different components, we suggest treating each component as a single entity in the roofline model for 1) simplicity and 2) capturing the actual performance of the component, proposing a component-based roofline. Without loss of generality, we take the mixed-precision computation in Cube as an example to illustrate the design of our analysis.

4.1 Component-based Roofline Model

Deriving the actual performance of Cube A_{cube} is trivial.

$$A_{\text{cube}} = \frac{O_{\text{cube}}}{T_{\text{total}}}, \tag{1}$$

where O_{cube} is the number of operations executed by Cube from profiling, and T_{total} is the total time the operator takes.

In contrast, determining the ideal performance of the component presents a challenge. Different precision instructions can unleash varying arithmetic power, with INT8 instructions boasting peak arithmetic twice that of FP16. Similarly, as described in Section 2.1, the bandwidth of transfers within the same MTE is not necessarily the same. We first consider two naive methods, which unfortunately don't work.

- **Maximum Arithmetic Power.** A straightforward solution is to use the maximum arithmetic power among the precision instructions supported by Cube as the ideal Cube performance, which suggests only leveraging the fastest instruction, i.e., lowest precision, to achieve the best performance. However, this is impractical as developers may want to leverage the higher precision instructions to achieve better accuracy.
- **Average Arithmetic Power.** Another solution is to use the average arithmetic power of all precision instructions in Cube as the ideal Cube performance. However, in some cases, the actual performance of Cube may be higher than this "ideal performance" when leveraging a higher percentage of the lowest precision instructions, e.g., 100% INT8 instructions.

Operator-aware Ideal Performance. According to the above analysis, we observe that the ideal performance of the component should be related to the operator's execution. For instance, the highly demanded precision instructions should contribute more to the ideal Cube performance. The easiest way is to use the time of each operation with different precisions as the weight to calculate the weighted average ideal arithmetic power. However, due to the lack of a timing mechanism for transfers on AICore hardware, it is not feasible to accurately capture the duration of each precision operation from profiling. In contrast, only the counts for each precision type of instruction are available, which can be obtained by profiling the instruction queue of each component.

Definition 1 (Ideal Cube Performance)**.** The operator-aware ideal performance of the Cube I_{cube} is the maximum performance, on the other hand, related to the minimum time (ideal time T_{ideal}) to process the mixed-precision operations, i.e.,

$$I_{\text{cube}} = \frac{O_{\text{cube}}}{T_{\text{ideal}}}. \tag{2}$$

As the ideal arithmetic power of each precision instruction P_{prec} and the number of operations of each precision O_{prec} are available, the ideal time can be obtained as follow:

$$T_{\text{ideal}} = \sum_{\text{prec}} \frac{O_{\text{prec}}}{P_{\text{prec}}}. \tag{3}$$

Subsequently, we obtain

$$I_{\text{cube}} = \frac{\sum_{\text{prec}} O_{\text{prec}}}{\sum_{\text{prec}} \frac{O_{\text{prec}}}{P_{\text{prec}}}}, \tag{4}$$

i.e., the operator-aware ideal performance of the Cube I_{cube} is the weighted harmonic mean of the performance of all precision instructions within the Cube by treating the operations of each precision instruction as the weight.

The harmonic mean makes sense in this context because we can treat each precision as a task, where the number of operations is the amount of work done by the task and the performance is the rate of work done. Intuitively, the harmonic mean weights the slower tasks more heavily, as they impose greater limits on the system.

Definition 2 (Cube Utilization)**.** The utilization of the Cube U_{cube} can be calculated as the ratio of the actual performance to the ideal performance, i.e.,

$$U_{\text{cube}} = \frac{A_{\text{cube}}}{I_{\text{cube}}}. \tag{5}$$

Insight 1. Due to hardware profiling frequency limitations, we can only obtain performance data for hardware units over a period of time, rather than their precise performance at each clock cycle. Taking the Cube unit as an example, we track the total number of cycles for operator execution (T_{total}) and the total number of Cube operations (O_{cube}) to calculate its actual compute performance (A_{cube}). Therefore, in the roofline model, we assess whether the hardware has reached its compute/bandwidth limits by evaluating the overall arithmetic/bandwidth utilization. For instance, if U_{cube} exceeds the practical threshold, we consider that the operator's computational power has reached the hardware limit.

4.2 Underutilization Analysis

In deployment, we set a threshold for the utilization of each component, and if the utilization exceeds the threshold, the component is considered a bottleneck (bound). In contrast, if the utilization of all components is below the threshold, the hardware is underutilized, and further analysis is required.

Utilization Decomposition. Fortunately, our detailed profiling data provides more available information, which can be further employed to identify the root cause of the underutilization of a component. An important metric is the execution time of a given component, which is the time spent on the component during the operator execution like T_{cube}. This metric provides a more detailed view of the performance of each component. For example, the utilization of the Cube

component can be transformed into the following formula:

$$U_{\text{cube}} = \frac{O_{\text{cube}}}{T_{\text{op}} \cdot I_{\text{cube}}} = \underbrace{\frac{O_{\text{cube}}}{T_{\text{cube}} \cdot I_{\text{cube}}}}_{E_{\text{cube}}} \cdot \underbrace{\frac{T_{\text{cube}}}{T_{\text{total}}}}_{R_{\text{cube}}}. \tag{6}$$

Eq. 6 decomposes the utilization of the Cube component into two parts: E_{cube} and R_{cube}. The E_{cube} measures the execution efficiency of the Cube component, which is the ratio of the actual execution performance to the ideal performance. The R_{cube} measures the proportion of time spent on the Cube component during the operator execution. Accordingly, the underutilization of the Cube component can be further attributed to the inefficiency when executing Cube operations or the insufficient time spent on the Cube component. Note that the same decomposition can be applied to other components.

The following insight gives further intuition into the design of our utilization analysis.

Insight 2. The Cube efficiency reflects the weighted average efficiency of all precision instructions in the Cube. Firstly, we obtain:

$$E_{\text{cube}} = \frac{\sum_{\text{prec}} \frac{O_{\text{prec}}}{P_{\text{prec}}}}{T_{\text{cube}}}. \tag{7}$$

For a given precision, the efficiency E_{prec} is calculated as:

$$E_{\text{prec}} = \frac{O_{\text{prec}}}{T_{\text{prec}} \cdot P_{\text{prec}}}, \tag{8}$$

where O_{prec} is the number of operations, T_{prec} is the execution time and P_{prec} is the peak performance for the given precision. Therefore, the Cube efficiency can be regarded as:

$$E_{\text{cube}} = \frac{\sum_{\text{prec}} E_{\text{prec}} \cdot T_{\text{prec}}}{\sum_{\text{prec}} T_{\text{prec}}}. \tag{9}$$

Classification of Underutilization. We observe that the time ratio of a given component can be affected by other components, as it reflects the component activity during the operator execution. In contrast, a component's efficiency is more intrinsic to the component itself, as it measures the performance when the component is active. Moreover, for an underutilized component, the high time ratio must result in low efficiency. Therefore, we use the time ratio as the core metric to identify the cause that leads to underutilization.

Recall the execution timeline of an operator in Figure 4b, the execution of different components can be overlapped to improve the overall performance. A good implementation that fully parallelizes the execution of different components has a bound component with a time ratio close to 1. Accordingly, if all components have low time ratios, it indicates that the operator execution is not fully parallelized and there still exists room for optimization. Formally, we have:

$$\forall \text{component}, R_{\text{component}} < R_{\text{threshold}} \Rightarrow \quad \textbf{Insufficient Parallelism}.$$

where $R_{\text{component}}$ is the time ratio of the component, $R_{\text{threshold}}$ is the threshold to determine whether it is fully parallelized. Otherwise, if it is, we have

$$\begin{aligned} &\exists \text{component}, R_{\text{component}} \geq R_{\text{threshold}} \\ \Rightarrow &\exists \text{component}, E_{\text{component}} \leq \frac{R_{\text{threshold}}}{U_{\text{threshold}}} \\ \Rightarrow &\exists \text{component}, \textbf{Inefficient Component}. \end{aligned}$$

Therefore, an inefficient component must exist, leading to underutilization. Moreover, we classify the inefficient component into *Inefficient MTE* and *Inefficient Compute* for the memory and compute components, respectively. In more detail, among different transfers within the same MTE, those with the largest number of bytes transferred are often the most likely to be problematic. Similarly, within the same computing unit, the proportion of instructions with different precision can aid in diagnosing the issue, as the most prevalent instructions often contribute directly to inefficiency.

Revisit of incorrect analysis of transfer in Figure 3a. Combining the transferred bytes of matrices A and B through transfers $GM \rightarrow L0A$ and $GM \rightarrow L0B$, respectively, the actual performance of the MTE-GM can achieve the ideal bandwidth, i.e., the peak bandwidth of $GM \rightarrow L0A/B$. According to the underutilization analysis, the MTE-GM will be identified as the bound bottleneck as it is fully utilized.

Revisit of incorrect analysis of compute unit in Figure 3b. According to the definition, the actual performance of the cube unit is 2/3 of the peak performance of the INT8 instruction. In contrast, the operator-aware ideal performance, maximum performance, and average performance of the cube unit are 2/3, 1, and 3/4 of the peak performance of the INT8 instruction, respectively. As we assume the Cube unit is fully utilized, i.e., 100% utilization, the operator-aware ideal performance perfectly matches the actual performance.

4.3 Pruning, Visualization, and Analysis

The component-based roofline model consolidates multiple transfers/precision operations within the same MTE/compute unit into a single component. However, this results in 3 compute components and 15 memory components (3 MTEs and 12 non-MTE transfers), totaling 45 combinations, which can still be overwhelming for visualization and analysis.

Pruning. The pruning can be conducted from two aspects: 1) the memory components that do not contribute to performance bottleneck and optimization, and 2) the impossible combinations of memory and compute components. After component abstraction and two prunings, the combinations drop from 180 to 7. Regarding the Ascend chip in Figure 1, we notice that the transfers directly connected to the compute unit, such as $L0A/B \rightarrow Cube$, are inevitable and leave no room for optimization, thus the corresponding analysis is unnecessary. Moreover, direct transfer between $L0C \rightarrow UB$ is rare, and Ascend implements this transfer through $L0C \rightarrow Vector$ and $Vector \rightarrow UB$. Overall, only

MTE should be considered. Moreover, the impossible combinations of MTEs and compute units, including (MTE-L1, Vector) and (MTE-L1, Scalar), are pruned from the analysis.
Visualization. The component-based roofline model is visualized in Figure 6, where the bandwidth ceilings of different MTEs and the arithmetic ceilings of various compute units are set. Accordingly, when analyzing bottlenecks, only a maximum of 7 performance points need to be considered. The utilization of each component is visualized by the closeness of the performance point to its corresponding ceiling.
Analysis. As shown in Figure 7a, we take the Add_ReLU operator to demonstrate how to analyze performance using the component-based roofline model. Since this operator only involves vector and scalar computations, along with MTE-GM and MTE-UB transfers, the figure shows only 4 performance points. The performance point closest to its ceiling (for Vector and MTE-UB) achieves the highest *MTE_utilization* of only 38.42%, indicating significant underutilization. Furthermore, within the pipeline formed by Scalar, Vector, MTE-GM, and MTE-UB instructions, the component with the highest *component_time_ratio* (MTE-GM) accounts for only 58.68%, indicating *insufficient parallelism*, which requires optimization by increasing inter-component parallelism.

5 Optimization Experience

Using the unique capabilities of Ascend hardware, we present three detailed operator optimization case studies in the inference of the MobileNetV3 model: Add_ReLU, Depthwise, and AvgPool. These case studies illustrate how we select optimizations based on the results of the roofline analysis.

5.1 Optimization of Add_ReLU Operator

In MobileNetV3, the Add_ReLU operator primarily appears in the Hard-Swish [21] activation. This fused operator combines the Add and ReLU operators to accelerate computation on Ascend and its formula is as follows:

$$\text{Add\_ReLU}(x) = \text{ReLU}(x + c), \tag{10}$$

where c is the constant defined by the activation function. As shown in Figure 8, the execution of the Add_ReLU operator can be divided into four steps: ① transfer the inputs (x and c) from GM to UB (MTE-GM); ② perform the Add operation, with the result stored in UB (Vector); ③ perform the ReLU operation, with the result stored in UB (Vector); ④ transfer the computed result from UB back to GM (MTE-UB).
Iteration 1: insufficient parallelism. We chose to optimize the Add_ReLU operator with the longest execution time (98.673 µs). According to the analysis in Section 4.3, it suffers from *insufficient parallelism*.
Reducing Spatial Dependency (RSD). Memory contention is a common problem that affects pipeline parallelism. A typical example is the bank conflict [48] in GPUs, where multiple threads access the same memory bank, allowing only one request to be processed while others must wait. In Ascend, this is reflected as instructions from different MTEs simultaneously accessing the same memory address, known as *spatial dependency*. In the Add_ReLU code, we identified a strong spatial dependency between MTE-GM and MTE-UB instructions. As shown in Figure 9, line 3 represents writing data back from 'ub_1' to 'gm_1' after the previous round of computation, while line 4 indicates transferring the next round of data from 'gm_2' to 'ub_1'. When the instructions are parallelized, both read and write access the same address, leading to memory contention and interrupting parallel execution. Therefore, if memory resources allow, it is best to allocate separate memory for computation results to reduce spatial dependency. After completing the previous round of computation, we can request a new memory 'ub_2' in UB to store the results, while the next input continues to load into 'ub_1'. With this optimization, line 9 and line 10 can execute in parallel, enabling pipelining across different rounds.
Iteration 2: MTE-UB bound. After applying RSD optimization, as shown in Figure 7b, the roofline analysis indicates that the performance point for Vector and MTE-UB has the highest *MTE_utilization*, reaching 66.24%, surpassing the bound threshold for vector operations. Notably, vector operations often run on smaller data blocks with frequent transfer requirements, which limits their utilization. Further analysis reveals that MTE-UB has the highest *component_time_ratio* at 85.14%, confirming that this operator is *MTE-UB bound*.
Minimizing Redundant Transfer (MRT). Apart from hardware bandwidth limitations, MTE bounds are most likely caused by redundant data transfers. In particular, transferring data that is independent of the loop variable inside a loop causes this, and it is generally recommended to move such data transfers out of the loop. For this operator, as illustrated in Figure 10, the original code attempts to transfer the constant c from GM to UB during each loop, even though this should only be done once. Therefore, moving the memory transfer of constant c outside the loop can alleviate the MTE Bound.
Overall Results. After applying MRT optimization, as shown in Figure 7c, the roofline analysis indicates that the performance point for Vector and MTE-UB remains the highest *MTE_utilization*. With reduced memory transfers, the *MTE_utilization* has improved to 70.52%, while the bottleneck remains *MTE-UB bound*. All optimizations reduce the operator's execution time from initial 98.673 µs to 57.157 µs. The overall inference latency is reduced by 244.261 us after optimization of all Add_ReLU operators. During the iterative analysis and optimization, we realize that a single round of optimization might not eliminate bottlenecks, and they might even shift to other parts. Therefore, continuous analysis and optimization of operators are imperative.

5.2 Optimization of Depthwise Operator

In MobileNetV3, the depthwise [30] operator is a modification of the standard convolution, performing independent calculations on each input channel to reduce computational

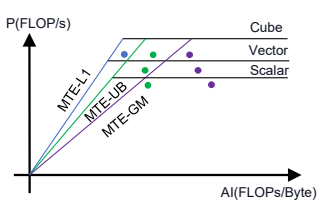
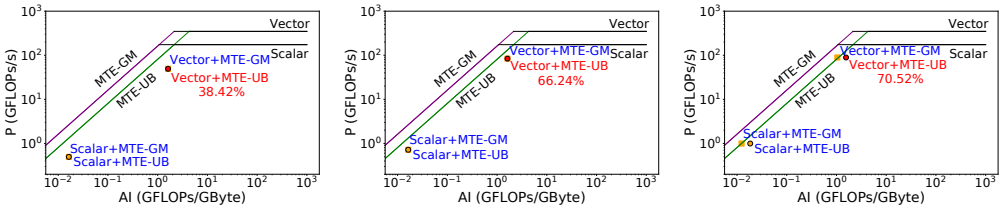
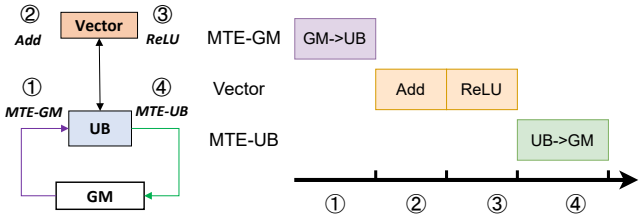


Figure 6. Component-based roofline.



(a) Before optimization. **(b)** Parallel optimization. **(c)** MTE-bound optimization.

Figure 7. Roofline of Add_ReLU operator optimization.



(a) Data flow. **(b)** Execution timeline.

Figure 8. The execution of Add_ReLU.

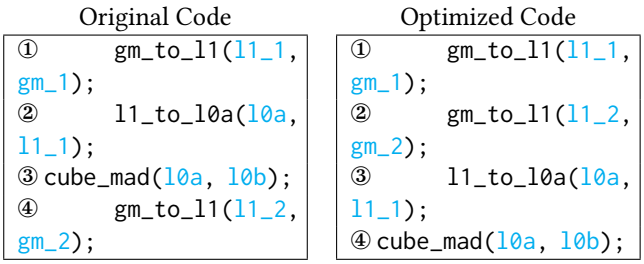


Figure 11. Code of AIS optimization.

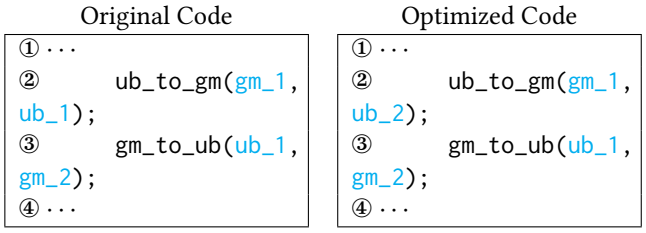


Figure 9. Reducing spatial dependency.

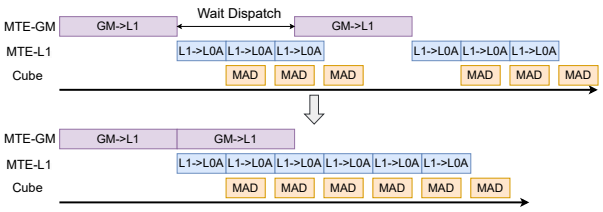


Figure 12. Adjusting instruction sequence.

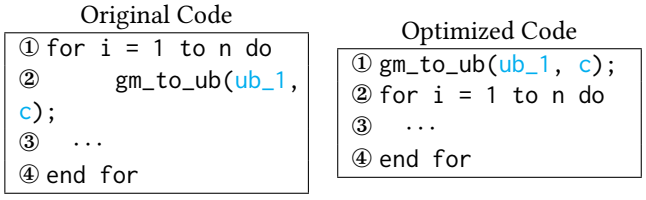


Figure 10. Minimizing redundant transfer.

load. Its core operation also involves multiplication and addition, with the formula as follows:

$$Y_{i,j} = \langle X_{[i:i+k,j:j+k]}, W \rangle = \text{sum}(X_{[i:i+k,j:j+k]} \circ W), \quad (11)$$

where $Y_{i,j}$ represents the output, $X_{[i:i+k,j:j+k]}$ refers to the input, and W denotes the weight. In the Ascend implementation, the operator typically includes the following steps: input data is transferred from GM to L1 (MTE-GM), then from L1 to L0A/B (MTE-L1), and finally, the Cube unit performs the multiply-add operations (Cube).

Iteration 1: insufficient parallelism. The depthwise operator with the longest execution time reaches 408.101 µs. A bottleneck analysis reveals that the *MTE_utilization* is only 35.45%, indicating a clear underutilization. Furthermore, analysis reveals that the most time-consuming component (MTE-GM) accounts for 46.66% of the total time, which is far below the threshold. This indicates a significant *insufficient parallelism*, likely requiring multiple optimizations.

Adjusting Instruction Sequence (AIS). As shown in Figure 11, lines ① and ④ in the original code correspond to two MTE-GM transfers, with multiple MTE-L1 transfers and MAD operations in between. Referring to the instruction queue in Figure 12, we observe a noticeable delay between the two MTE-GM transfers. This delay occurs because the AI Core sequentially retrieves instructions and dispatches them to the respective instruction queues, with the dispatch overhead increasing as the number of intermediate instructions grows. In contrast, the optimized code allows the MTE-GM transfer in line ② to proceed without waiting, thereby achieving better overlap with MTE-L1 transfers and Cube computations.

Removing Unnecessary Synchronization (RUS). After the AIS optimization, the *MTE_utilization* increases to 42.72%, but still remains underutilization. The component with the highest time ratio is MTE-L1, which reaches 71.69%, still indicating insufficient parallelism. Upon reviewing the operator code, we find excessive '*pipe_barrier* (PIPE_ALL)' instructions between MTE-L1 and MTE-UB transfers, forcing all components to synchronize and causing instructions to execute sequentially. Removing unnecessary synchronization instructions can help improve component parallelism.
Ping-pong Policy (PP) [34]. After the RUS optimization, the operator's *MTE_utilization* improves to 60.35% but still indicates underutilization for cube operations. The highest *component_time_ratio* is 77.69%, leaving room for improving parallelism. By examining the code, we identified that the ping-pong policy was not employed, if a memory unit is already occupied, concurrent read and write operations are no longer supported. For example, with the L1 buffer, the $GM \rightarrow L1$ transfer must wait for the $L1 \rightarrow L0A$ transfer to complete before it starts, limiting component parallelism. The ping-pong strategy divides the memory unit into two sections: during a clock cycle, one section handles reading while the other handles writing. In the next cycle, their roles swap. Applying this reduced the MTE-GM waiting intervals from 14 to 3, significantly improving parallelism.
Iteration 2: inefficient MTE. After three parallel optimizations, *MTE_utilization* increased from 35.45% to 71.56%. Despite the underutilization, the *component_time_ratio* of the MTE-GM reached 94.18%, indicating that the cause has shifted from insufficient parallelism to *inefficient MTE-GM.*
Increasing Transfer Granularity (ITG). In our experience, the most common cause of inefficient MTE is overly small transfer granularity, which increases the overhead of maintaining transfer connections and leads to low bandwidth utilization. Our modeling and analysis of various transfer links also confirm this finding. By examining the granularity of MTE-UB transfers, we find that each $UB \rightarrow GM$ transfer is only 30 KB, far below the threshold for full bandwidth. Therefore, we can merge multiple transfers to increase the granularity of MTE-UB transfer, improving bandwidth utilization.
Overall Results. After all optimizations, the depthwise operator finally achieves MTE-GM Bound with the utilization of 93.54% and reduces from 408.101 µs to 325.121 µs. The overall latency is reduced by 347.513 µs after optimizing all depthwise operators. From the complex parallel optimization, we find that optimizing the bottleneck is not a one-step process; rather, it often requires addressing multiple interrelated issues. This requires combining instruction pipelining with operator code inspection to systematically identify problematic instructions and source code.

5.3 Optimization of AvgPool Operator

In MobileNetV3, the AvgPool operator generates the output by computing the average of each pooling window of the input feature map on Vector. Its formula is as follows:

$$Y_{i,j} = \frac{1}{k^2} \cdot \text{sum}(X_{[i:i+k,j:j+k]}), \tag{12}$$

where $Y_{i,j}$ represents the output, $X_{[i:i+k,j:j+k]}$ represents elements in the pooling window, and k^2 is the size of window.
Iteration 1: inefficient compute. By the bottleneck analysis of a typical AvgPool operator, we find that its utilization is only 13.54%, indicating underutilization. Further analysis reveals that the Vector accounts for the largest ratio of time, reaching 83.98%. This pipeline parallelism is relatively high, but there is a *inefficient compute* bottleneck in the Vector.
Adjusting Instruction Parameter (AIP). The inefficient computing units often arise from *unreasonable parameter settings* in operator implementations. For instance, the *repeat* parameter, which controls the number of times an operation is repeated, allows hardware to perform repeated executions without explicit loops. When this parameter is too low, additional loops and data transfers are required to compensate, leading to inefficient computation. Similarly, the *mask* parameter determines which elements are activated during a computation, and can hinder efficiency if set too low, as multiple instructions may be needed to complete. In this AvgPool operator code, the *repeat* parameter for the Add operation is only 1, resulting in 98 loops. Thus, we can eliminate these extra loops by increasing the *repeat* parameter to 98.
Overall Results. After AIP optimization, the Vector utilization improved to 59.07%, and the time decreased from 69.821 µs to 16.206 µs. Although this is still far from the threshold, it represents the upper limit of software optimization for lightweight models. After optimizing all AvgPool operators, the total time decreased by 447.923 µs.

5.4 Summary

In addition to the common underutilization bottleneck optimizations described above, we also summarize some optimizations for MTE and compute-bound bottlenecks.
MTE Bound. To mitigate the limited bandwidth in MTE-bound operators, we usually adopt the following strategies:

- *Operator Fusion (OP) [8, 35].* Some operators can be fused into a single operator to reduce memory access, such as operators integrating BatchMatmul and Add. And cache strategies, like FlashAttention [14], can reduce data transfers by directly computing the output.
- *Transfer Transformation (TT).* Another method is to leverage the various data transfers with significantly different bandwidths. By switching the transfers to paths with higher bandwidth, memory-bound can be alleviated.

Compute Bound. Similarly, the optimization of compute bound can be approached from the following perspectives:

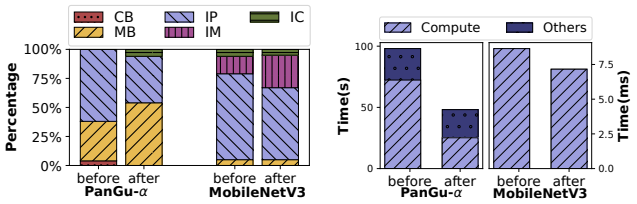
- *Enhanced Algorithm (EA).* To tackle compute-bound issues, more efficient algorithms are often employed. For

Table 1. Optimization and speedup of operators.

Operator	Performance Impediment and Optimization Strategy					Speedup
	Compute Bound	MTE Bound	Insufficient Parallelism	Inefficient MTE	Inefficient Compute	
Add_ReLU [28]		MRT	RSD			1.72
Depthwise [30]		MRT	AIS,RUS,PP	ITG		1.26
AvgPool [29]					AIP	4.31
Mul [32]			RSD			1.34
Conv2D [1]		MRT	RSD			2.65
FullyConnection [37]				ITG		1.22
MatMul [2]		OP				1.10
GeLU [31]	EA					1.06

Table 2. Workload specification

Type	Model	Parameter	Dataset	#NPUs
Vision	MobileNetV3(M3)	5.4M	ImageNet2012	8
	ResNet50	25.6M		
	ViT	86M		
	VGG16	138.4M		
NLP	Bert	110M	WikiText2	8
	GPT2	355M		
Recommendation	DeepFM	16.5M	Criteo	8
	Wide and Deep(W&D)	75.84M		
	DLRM	540M		
LLM	Llama 2	7B	WikiText2	8
	PanGu-α	100B	1.1TB Chinese Dataset	128



(a) Bottleneck cause. **(b)** Execution time.

Figure 13. Optimization results (CB, MB, IP, IM, and IC denote Compute Bound, MTE Bound, Insufficient Parallelism, Inefficient MTE, and Inefficient Compute).

instance, depthwise separable convolution [12] and improvements in activation functions, such as FastGeLU [58], can significantly reduce computational cost.

- *Low-precision Calculation (LC).* We can leverage the quantization algorithms [27, 44, 51] to transform the high-precision computation to low-precision computation for better processing performance.
- *Computation Transformation (CT).* The Ascend chip's three computing units exhibit increasing computing power. When a lower-performing compute unit reaches its limit, the computation can be shifted to a more powerful compute unit through data rearrangement [26, 42], and compilation optimization [10, 52, 57].

Operator performance varies significantly across different models. However, for some universal operators, their performance bottlenecks tend to be consistent. Therefore, as shown in Table 1, we summarize the performance impediments of these presentative operators in MobileNetV3 and propose targeted optimization suggestions.

6 Evaluation

6.1 Experimental Setting

Hardware. All experiments were conducted on Ascend chips, including the Ascend inference [4] and training chip [5]. The inference deployment was executed on a single card, while the training was deployed on the Peng Cheng Cloud Brain II cluster [43], with each node comprising eight cards.
Workloads. In our experiments, we examined the workloads from various models, encompassing over a dozen models such as NLP, vision, recommendation, and large language models (LLMs). Table 2 details their specifications for training. Additionally, to validate the effectiveness of optimization across different Ascend hardware, we selected a subset of these models for inference on the Ascend inference chip. Notably, these workloads are based on state-of-the-art models officially provided by MindSpore [33], each refined through four years of continuous optimization.
Roadmap of Evaluation. Initially, we present two end-to-end studies on the training of 100B PanGu-α model [56] and the inference of MobileNetV3 model [21] to demonstrate how to employ our approach to optimize real-world workloads. Then, a comprehensive analysis and optimization among various models are presented to showcase the generality and effectiveness of our approach, followed by a discussion of the insights gained from these experiments.

6.2 End-to-end Optimization

We present two end-to-end studies to demonstrate how to leverage our analysis and optimization strategies to enhance the performance of real-world workloads.

6.2.1 Case 1: PanGu-α Training. LLMs, powered by immense parameters and datasets, have witnessed significant enhancements in their capabilities, yet this comes with significant training time costs [9]. Based on our practical experience, the computation time often accounts for over 50% of the total training time in LLMs, highlighting the importance of analyzing and optimizing these computation operators.
An overview of performance impediments. After profiling the 100B PanGu-α model training, we analyze each operator's performance within an iteration by the enhanced roofline model. As shown in Figure 13a, our analysis revealed that operators with Insufficient Parallelism accounted for the largest share of 61.48%, followed by MTE-Bound operators at 34.02%, while Compute-Bound operators only comprised 4.50%. Among the MTE-Bound operators, approximately 90.30% were found to be bound by MTE-GM bandwidth. This assessment suggests that low parallelism between components is the primary challenge, coupled with a significant bandwidth bottleneck arising from MTE-GM.
Detailed analysis and optimizations. In practice, we prioritize operator optimizations based on execution time, with

longer-running operators receiving higher priority. We selected the top 10 longest-running operators for optimization, focusing on alleviating the insufficient parallelism bottleneck they exhibit. Briefly, 10 types of operators can be summarized as matrix multiplication operators (MatMul, BatchMatMul), format conversion operators (e.g., TransData and Cast), activation operators (GeLU, DropoutDoMask), and element-wise operators (e.g., Mul, Add, AddN, and RealDiv). In a single iteration, they collectively consume 59.59 s, accounting for 83.57% of the total computation time. Thus, the greatest potential gains can be achieved by addressing the root causes of bottlenecks with the following optimizations:

- We identified many element-wise operators such as Mul, Add, AddN, and RealDiv suffer from insufficient parallelism. These operators can be optimized by fusing them into a LayerNorm operator for higher parallelism.
- For inefficient activation operators like GeLU and DropoutDoMask, we substituted them with high-performance ones, FastGeLU and DropoutDoMaskV3.
- For MTE-bound matrix multiplication operators, we proposed fusing MatMul with FastGeLU and BatchMatmul with Add to reduce the cost of memory access.
- The inefficient format conversion operators stem from the Cube Unit's requirement to convert input tensors from arbitrary formats into the private format [6] for efficient processing. We can minimize their occurrence by adjusting the input format.

As shown in Figure 13b, after these optimizations, the computation time is significantly reduced from 72.31s to 25.16s and the iteration time fell from 98.01s to 48.16s. While these optimizations have brought positive changes, the distribution of operator bottleneck causes has undergone remarkable changes. Notably, the percentage of operators suffering from Insufficient Parallelism dropped significantly to 40.10%. Conversely, as parallelism issues eased, many operator bottlenecks shifted to MTE-Bound, with their proportion increasing to 53.45%. Moreover, 47.37% of operators are MTE-GM bound, all originating from the $GM \rightarrow UB$ transfer, which is difficult to alleviate through software optimizations. This insight highlights the insufficient bandwidth in current LLM training, emphasizing the need of next-generation chips.

6.2.2 Case 2: MobileNetV3 Inference. Compared to the PanGu-α LLM, MobileNetV3 is a lightweight CNN that significantly reduces the model size and computational load, making it more suitable for resource-constrained environments. We conduct experiments on MobileNetV3 to validate the generality of our approach on Ascend inference chips.

Compared to training, operator bottlenecks in inference show more underutilization, with more inefficient components emerging, particularly within MTE transfers. As shown in Figure 13, among the 155 computation operators, the most significant bottleneck was Insufficient Parallelism (73.55%), followed by Inefficient MTE (15.48%), while the rest were Inefficient Compute (6.45%) and MTE Bound (4.52%). Table 1 summarizes the optimization and results for the time-consuming operators. After optimization, the total time decreased from 8642 µs to 7157 µs and the distribution of the bottleneck causes underwent significant changes. The insufficient parallelism has significantly decreased to 61.94%, while the Inefficient MTE has risen to 28.39%. The inefficient compute slightly dropped to 4.52%, while the MTE-Bound has increased to 5.16%.

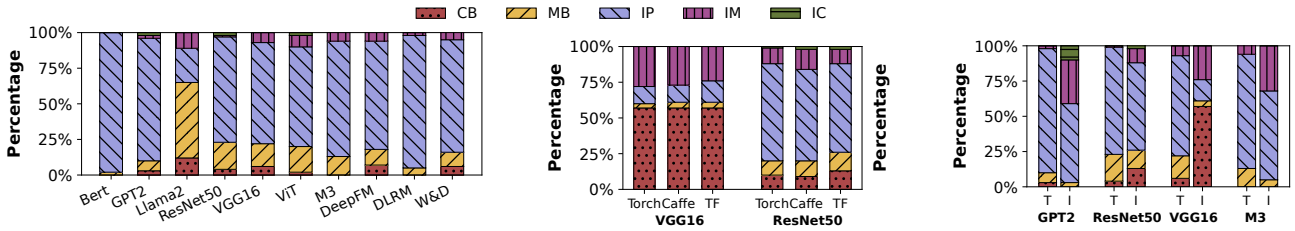
6.3 Insights from Comprehensive Experiments

In addition to the models discussed, we conduct a comprehensive study of operator optimization, further validating the breadth of the optimizations and gaining valuable insights.

Training Various Models. As depicted in the causes of training bottlenecks in Figure 14a, *the distribution of performance impediments varies significantly across different models.* We can further deduce that the model parameter size is a key factor affecting the causes of bottlenecks. For small models, the main issue is insufficient parallelism, rarely reaching compute or MTE Bound. However, due to their substantial computation and data transfer demands, large models like 7B Llama2 and 100B PanGu-α, are particularly prone to component bound, especially MTE-GM bound. This is primarily driven by the frequent $GM \rightarrow UB$ transfers required in numerous vector computations, compounded by relatively limited GM read bandwidth. Additionally, we observe that, aside from the Llama2 model, other models exhibit significant IP bottlenecks, indicating suboptimal operator implementations and suggesting considerable room for optimization.

Various Programming Frameworks. We also analyze the impact of different programming frameworks on the performance impediments of the same model. The Ascend inference chip enables the conversion of models from mainstream DL frameworks such as TensorFlow, PyTorch, Caffe, and into executable formats for Ascend processors. This allows us to evaluate the bottleneck distribution of the same model across different frameworks. As shown in Figure 14b, take inference for example, we reveal that *the programming framework has little impact on the performance impediments*, owing to the same operator library on the Ascend hardware. This also indirectly reveals that our operator optimizations could be effectively applied across different frameworks.

Training vs. Inference. As shown in Figure 14c, we also explore the differences and correlations of bottleneck causes between training and inference workloads, including GPT-2, MobileNetV3, ResNet50, and VGG16 models, using different Ascend chips for training and inference, respectively. Briefly, *the impediment causes differ significantly for the same model under different workloads.* Specifically, insufficient parallelism is common, underscoring the need for optimized operator implementations. In models with more efficient implementations, like ResNet50 and VGG16, the inference



(a) Different models. **(b)** Different frameworks. **(c)** Inference vs. training.

Figure 14. The distribution of performance impediments.

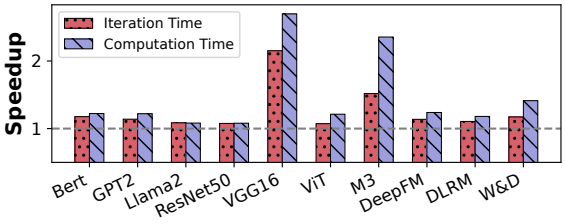


Figure 15. Time speedup with optimization.

chip's lower compute capacity compared to the training chip is more likely to reach Compute Bound. Additionally, as data transfer requirements grow substantially during training, training workloads are more prone to MTE Bound, particularly the memory bandwidth in MTE-GM and MTE-UB, which calls for improvements in future chip designs. Conversely, reduced computation and data transfer in inference tend to inefficient compute and MTE units.

Optimization. Guided by the bottleneck analysis across various models, we accelerate their training by applying operator optimization techniques discussed in Section 5. For example, we optimize the widespread insufficient parallelism by adjusting instruction sequence, reducing spatial dependency, and using the ping-pong policy. And for models like Llama2 that suffer from severe MTE Bound, we mitigate these by minimizing redundant transfers. The corresponding speedups are illustrated in Figure 15, including the operator computation time and the overall time, which further involves communication and I/O in a single iteration. It is evident that through operator optimization, the operator computation time for each model decreased significantly, with speedups ranging from 1.08× to 2.70×. Consequently, the overall execution time also decreased to a certain extent. However, due to the influence of factors such as data preprocessing and communication, the speedup ratio is slightly lower, ranging from 1.07× to 2.15×. Nonetheless, the results demonstrate the effectiveness of our operator optimization.

7 Discussion and Future Work

Extend to other DSAs. Although the component-based roofline model was derived from the Ascend architecture, its methodology is also applicable to other DSAs. For example, Google TPU v5 [15] also undergoes improvements in memory and compute organization. It also incorporates heterogeneous compute units, including Matrix Multiply, Vector, and Scalar Units. For memory access, the Matrix Multiply Unit has two distinct memory paths, namely inputs from the Unified Buffer and the Weight FIFO, with significantly different bandwidths. These characteristics can be well represented by the concept of components, and the component-based roofline model can be extended to more DSAs in the future.

Detailed analysis of hardware. Due to profiling limitations on hardware, this paper does not fully explain how software implementations can cause hardware-related issues, such as bank conflicts [16], lack of double buffering [50], and improper memory tiling [23]. Therefore, we plan to explore the hardware architecture and its interaction with the software stack, including cycle-by-cycle analysis, detailed descriptions of hardware operations, and how optimizations translate into hardware efficiency.

8 Conclusion

The rise of deep learning has driven the development of Domain-Specific Accelerators (DSAs), whose unique architectures lead to a lack of effective performance analysis and optimization. This paper introduces a component-based roofline model for performance analysis, tailored to the architectural features of the representative DSA, Ascend. Through in-depth operator optimization case studies, we guide users on how to use the enhanced roofline model for optimization. Additionally, we share insights and lessons from extensive optimization experiments in production environments, hoping to inspire future DSA designs.

Acknowledgments

The authors thank the anonymous reviewers for their insightful comments. This work was supported by the Key Program of the Natural Science Foundation of Jiangsu Province under Grant No. BK20243053, the National Natural Science Foundation of China under Grant Numbers 62325205 and 62172204, and the Nanjing University-China Mobile Communications Group Co., Ltd. Joint Institute.

References

[1] Ascend. Conv2d. https://gitee.com/ascend/samples/blob/master/cplusplus/level1_single_api/4_op_dev/1_custom_op/doc/Conv2d_EN.md.

[2] Ascend. Matmul. https://gitee.com/ascend/samples/blob/master/cplusplus/level1_single_api/4_op_dev/1_custom_op/doc/Matmul_EN.md.

[3] Ascend. Profiling tool. https://support.huawei.com/enterprise/en/doc/EDOC1100269271/87e7e12.

[4] Ascend. Atlas 300i inference card. Website, 2024. https://e.huawei.com/en/products/computing/ascend/atlas-300-ai.

[5] Ascend. Atlas 300t training card. Website, 2024. https://e.huawei.com/en/products/computing/ascend/atlas-300t-training-9000.

[6] Ascend. Format. Website, 2024. https://www.hiascend.com/document/detail/en/CANNCommunityEdition/600alphaX/opdevg/opdevg/atlasopdev_10_0007.html.

[7] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE access*, 6:64270–64277, 2018.

[8] Xuyi Cai, Ying Wang, and Lei Zhang. Optimus: An operator fusion framework for deep neural networks. *ACM Transactions on Embedded Computing Systems*, 22(1):1–26, 2022.

[9] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.

[11] Zhengbo Chen, Fang Zheng, Qi Yu, Rujun Sun, Feng Guo, and Zuoning Chen. Evaluating performance of ai operators using roofline model. *Applied Intelligence*, 52(7):7054–7069, 2022.

[12] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[13] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[14] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[15] Google. Tpu v5e. Website, 2023. https://www.semianalysis.com/p/tpuv5e-the-new-benchmark-in-cost.

[16] Chunyang Gou and Georgi N Gaydadjiev. Elastic pipeline: addressing gpu on-chip shared memory bank conflicts. In *Proceedings of the 8th ACM international conference on computing frontiers*, pages 1–11, 2011.

[17] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[18] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. Sparse-tpu: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM international conference on supercomputing*, pages 1–12, 2020.

[19] Intel. Intel advisor roofline. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html, 2018.

[20] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[21] Brett Koonce and Brett Koonce. Mobilenetv3. *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, pages 125–144, 2021.

[22] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. Domain-specific architectures: Research problems and promising approaches. *ACM Transactions on Embedded Computing Systems*, 22(2):1–26, 2023.

[23] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing 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)*, pages 789–801, 2021.

[24] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *Hot Chips Symposium*, pages 1–44, 2019.

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[26] Scott Lloyd and Maya Gokhale. In-memory data rearrangement for irregular, data-intensive computing. *Computer*, 48(8):18–25, 2015.

[27] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[28] MindSpore. Add. https://www.mindspore.cn/docs/en/master/api_python/ops/mindspore.ops.add.html.

[29] MindSpore. Avgpool. https://www.mindspore.cn/docs/en/master/api_python/ops/mindspore.ops.avg_pool1d.html.

[30] MindSpore. Depthwise. https://www.mindspore.cn/docs/en/r2.3.0rc1/api_python/ops/mindspore.ops.conv1d.html.

[31] MindSpore. Gelu. https://www.mindspore.cn/docs/en/master/api_python/ops/mindspore.ops.gelu.html#mindspore.ops.gelu.

[32] MindSpore. Mul. https://www.mindspore.cn/docs/en/master/api_python/ops/mindspore.ops.mul.html#mindspore.ops.mul.

[33] mindspore. Mindspore modelzoo. https://gitee.com/mindspore/models, 2024.

[34] Akira Nakamura, Keiko Haga, and Kunio Yamane. The transglycosylation reaction of cyclodextrin glucanotransferase is operated by a ping-pong mechanism. *FEBS letters*, 337(1):66–70, 1994.

[35] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson.

The design process for google's training chips: Tpuv2 and tpuv3. *IEEE Micro*, 41(2):56–63, 2021.

[37] NVIDIA. Linear/fully-connected layers user's guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html.

[38] NVIDIA. Nvidia nsight compute. https://developer.nvidia.com/nsight-compute.

[39] Pytorch. Pytroch profiler, 2024. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[40] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators. In *2019 IEEE high performance extreme computing conference (HPEC)*, pages 1–9. IEEE, 2019.

[41] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–12. IEEE, 2020.

[42] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362, 2010.

[43] Yun Su, Jipeng Zhou, Jiangyong Ying, Mingyao Zhou, and Bin Zhou. Computing infrastructure construction and optimization for high-performance computing and artificial intelligence. *CCF Transactions on High Performance Computing*, pages 1–13, 2021.

[44] Stefan Uhlich, Lukas Mauch, Fabien Cardinaux, Kazuki Yoshiyama, Javier Alonso Garcia, Stephen Tiedemann, Thomas Kemp, and Akira Nakamura. Mixed precision dnns: All you need is a good parametrization. *arXiv preprint arXiv:1905.11452*, 2019.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[46] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

[47] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009.

[48] Steve Worley. Optimization of primality testing methods by gpu evolutionary search. *GPUs for Genetic and Evolutionary Computation*, 2009, 2009.

[49] Ruohan Wu, Mingfan Li, Hanxi Li, Tianxiang Chen, Xinghui Tian, Xiaoxin Xu, Bin Zhou, Junshi Chen, and Hong An. Machine learning-enabled performance model for dnn applications and ai accelerator. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 25–34, 2022.

[50] Yang Xiao and Zeke Wang. Aibench: a tool for benchmarking huawei ascend ai processors. *CCF Transactions on High Performance Computing*, 6(2):115–129, 2024.

[51] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. Deep neural network compression with single and multiple level quantization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[52] Zhiying Xu, Jiafan Xu, Hongding Peng, Wei Wang, Xiaoliang Wang, Haoran Wan, Haipeng Dai, Yixu Xu, Hao Cheng, Kun Wang, and Guihai Chen. Alt: Breaking the wall between data layout and loop optimizations for deep learning compilation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 199–214, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronaghi, Adedoyin Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, Jack Deslippe, and Samuel Williams. An empirical roofline methodology for quantitatively assessing performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–23, 2018.

[54] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system. *Concurrency and Computation: Practice and Experience*, 32, 2019.

[55] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 503–521. USENIX Association, 2021.

[56] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, et al. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *arXiv preprint arXiv:2104.12369*, 2021.

[57] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.

[58] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.