CCF Transactions on High Performance Computing (2024) 6:115–129
https://doi.org/10.1007/s42514-024-00187-x

REGULAR PAPER

# AIbench: a tool for benchmarking Huawei ascend AI processors

Yang Xiao[1] · Zeke Wang[1]

Received: 27 June 2023 / Accepted: 27 March 2024 / Published online: 2 May 2024

**Abstract**
In recent years, plenty of AI accelerators, e.g., Google TPU and Huawei Ascend, have been proposed to accelerate various Deep Learning applications, such as CNN and NLP, because AI accelerators are specialized for AI model training and inference and can thus provide higher performance per watt than GPUs. Despite the wide adoption of AI processors in the deep learning domain, the potential of AI processors is not fully harvested in the other compute-intensive domains that need massive matrix and vector operations, because AI processors typically provide custom matrix and vector instructions. A significant challenge in harnessing AI processors in other domains is the undisclosed performance characteristics of these processors. To this end, we intend to benchmark AI processors in a comprehensive approach such that programmers can easily understand the performance characteristics of AI processors that always have similar architecture. Given this, we present AIBench, a benchmarking tool designed to reveal the underlying details of an AI processor. Initially, we benchmark Huawei's Ascend accelerator. The benchmarking results show (1) an Ascend 910 AI chip can provide 216 TFLOPs for float16 data from the matrix unit and 3390 GFLOPs from the vector unit and (2) the performance of an AI core is contingent upon the appropriate data transmission and operation mode. Utilizing unsuitable transmission modes can lead to data entry and exit times becoming a bottleneck.

**Keywords** Benchmarking · Domain-specific architecture · AI processor · Ascend

## 1 Introduction

Owing to the extensive application of deep learning, which requires substantial MACs for training and inference, general-purpose central processing units (CPUs) fail to deliver adequate computing power. This is primarily because CPUs are not optimized for compute-intensive tasks like deep learning, but are designed for versatility to run a broad range of applications. Consequently, graphic processing units (GPUs) are employed to expedite deep learning model training and inference, given their architecture featuring a plethora of computing units to accelerate parallel tasks. However, GPUs are not specifically designed for deep learning applications, their use in such tasks can result in elevated energy consumption. Conversely, AI accelerators, purpose-built for AI model training and inference, can deliver superior performance per watt compared to GPUs (Liao et al. 2021; Jouppi et al. 2017; Norrie et al. 2021; Jouppi et al. 2023; Liao et al. 2019). Despite the wide adoption of AI processors in the deep learning domain, their potential remains underutilized in other compute-intensive applications that require extensive matrix and vector operations. This is primarily due to the custom matrix and vector instructions typically provided by AI processors. However, the main challenge of leveraging AI processors in other domains is that the performance characteristics of AI processors have not yet been revealed. To close this gap, we intend to benchmark AI processors in a comprehensive approach enabling programmers to readily understand the performance characteristics of AI processors, which often have similar architecture between different vendors.

To this end, we present AIBench, a benchmarking tool that allows us to demystify all the underlying details of an AI processor. As a start, we benchmark Huawei's Ascend accelerator. The benchmarking results show that (1) an Ascend 910 AI chip can provide 216 TFLOPs for float16 data from

✉ Zeke Wang
wangzeke@zju.edu.cn

Yang Xiao
12221061@zju.edu.cn

1 Department, Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, Hangzhou 310027, Zhejiang, China

the matrix unit and 3390 GFLOPs from the vector unit and (2) the performance of an AI core depends on appropriate data transmission and operation mode. Employing inappropriate transmission modes may result in data ingress and egress times becoming a bottleneck. The performance of the AI core depends on appropriate data transmission and operation modes. Like Ascend accelerator, other AI processors also have a similar computational architecture that comprises matrix units and vector units. Consequently, it is easy to generalize AIBench to these AI processors that exhibit a similar computational pattern when executing compute-intensive tasks. Data openly available in a public repository. The code that used by this study are openly available in Github at https://github.com/Tinkerver/AIBench.

Specifically, the following main parts were carried out in this work:

- Evaluate the performance of each computational unit in the AI processor.
- Quantify the data transfer bandwidth among buffers within the chip and get the access patterns of these buffers.
- Investigate the caching mechanism of global memory and the interaction rate between global memory and on-chip buffers under various transfer modes.
- Evaluate the performance of AI processors under matrix and vector-intensive computational loads and assess the effectiveness of various optimization options.

## 2 Background

### 2.1 Ascend AI processor architecture

As depicted in Fig. 1, the Ascend processor architecture incorporates numerous AI cores (two AI cores in Ascend 310 chip and 32 AI cores in Ascend 910), which access a memory subsystem composed of HBM and DDR.

AIcore's computational units comprise a matrix unit, a vector unit, and a scalar unit, which perform the primary computational tasks of the Ascend processor. The matrix unit, serving as the computational powerhouse of the AIcore, swiftly executes matrix multiplication operations. The vector unit facilitates flexible vector operations in a SIMD manner. The scalar unit, responsible for executing scalar operations within the AIcore, handles tasks such as program loop control and address computation.

Surrounding the computational unit, DaVinci architecture establishes multiple on-chip buffers (including the unified buffer, L1 buffer, buffer L0A, buffer L0B, and buffer L0C). These buffers collaborate with the data path in the AI Core, facilitating input provision and output transmission for the computational unit.

Among these buffers, L0A/L0B/L0C are linked to the matrix computing unit. The L0A buffer houses the left matrix data for matrix multiplication, the L0B buffer accommodates the right matrix data for matrix multiplication, and the L0C buffer stores the results and intermediate outcomes

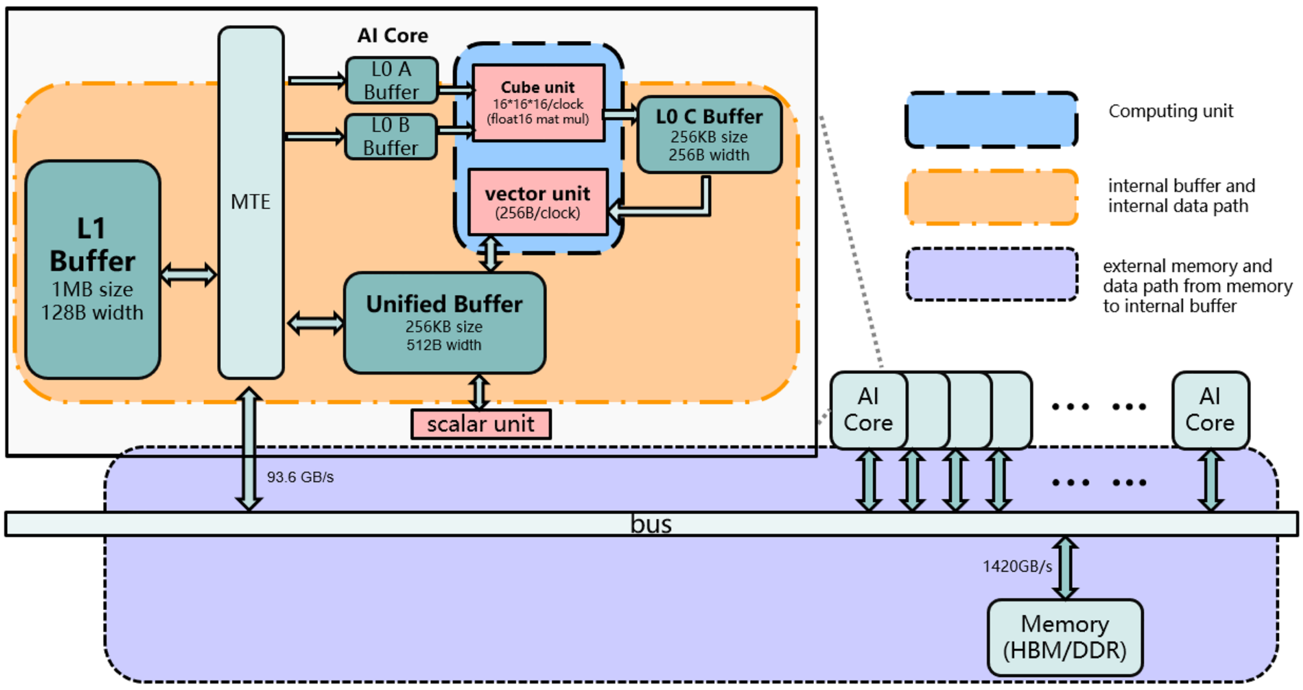

AI Core
MTE
L0 A Buffer
L0 B Buffer
Cube unit 16*16*16/clock (float16 mat mul)
L0 C Buffer 256KB size 256B width
vector unit (256B/clock)
L1 Buffer 1MB size 128B width
Unified Buffer 256KB size 512B width
scalar unit
Computing unit
internal buffer and internal data path
external memory and data path from memory to internal buffer
AI Core
AI Core
93.6 GB/s
bus
1420GB/s
Memory (HBM/DDR)

**Fig. 1** Ascend AI accelerator logical architecture

of matrix multiplication. The unified buffer is connected to the vector computing unit, and functions as the input and output location for the vector computational unit. The L1 buffer acts as the data transfer zone within AICore, temporarily holding certain data that AICore needs to reuse, thereby minimizing bus read and write operations.

During the benchmarking process of the computational units within AI cores, our attention will be centered on the cube processing unit and the vector processing unit, given their role as the main sources of computational power. The crux lies in examining the pipeline throughput of its arithmetic pipelines. About matrix units, we will further delve into the disparities corresponding to different shapes of matrix multiplication operational modes.

For the benchmarking of the internal buffer and data path, we measure the storage width of each buffer and the data transfer bandwidth between interconnected buffers. Additionally, we will assess the corresponding impact on bandwidth variations when transmissions occur with differing step sizes and iteration counts.

For the input of off-chip data, we examine the corresponding caching mechanism from memory to the AI chip and measure the peak bandwidth from memory to the AI core in both single-core and multi-core configurations.

Additionally, pertinent tests will be conducted on the functionalities of double buffering and barrier synchronization.

## 2.2 Programming APIs for TIK operators

TIK is a dynamic programming framework based on the Python language, presented as a Python module. Through the API provided by Tik, custom operators can be crafted based on Python, which is subsequently compiled into applications for the Ascend AI processor by the TIK compiler. Tik operates on Ascend AI acceleration chips, enabling precise control over data movement and computational processes within the AI core.

TIK's vector operations utilize single instruction multiple data (SIMD) instructions, with the fundamental operation unit in the instruction being distributed across two dimensions: space and time. Spatially, it is organized into blocks, while temporally, iteration is achieved by setting the 'repeat_times' parameter.

The API for single-vector operations within TIK is presented in the following format, with the specific parameter configurations detailed in Table 1.

***`Instruction(mask,dst,src,repeat_times,dst_rep_stride,src_rep_stride)`***

In TIK, the offset of the same block in adjacent repeats only supports linear mode, implying that users are required to specify the address offset for each block in the subsequent repeat. The principal operands of TIK's SIMD instructions are tensors, with a minority of operands being scalars (scalar/immediate).

## 2.3 Profiling tools

To scrutinize operator processes, we employ the msprof tool, a feature of Ascend CANN (compute architecture for neural networks), to amass performance profiling data within the operational environment. This encompasses the execution duration of matrix, vector, and scalar computational units, in addition to the corresponding time for "mte1" type instructions (L1 to L0A/L0B transfer instructions), "mte2" type instructions (DDR to AICORE transfer instructions), "mte3" type instructions (AICORE to DDR transfer instructions).

## 3 Overview of AIBench

We present AIBench, a benchmarking tool that enables us to unveal the intricate details of an AI processor, such as Huawei's Ascend accelerator. Consequently, programmers can accelerate other compute-intensive applications on AI accelerators if possible. In the following, we benchmark the computing unit of AI processors (Sect. 4), the on-chip memory subsystem of AI processors (Sect. 5), and the global memory subsystem of AI processors (Sect. 6). Then, we scrutinize the performance of computationally demanding workloads and the efficacy of conventional optimization techniques (Sect. 7). Subsequently, we deliberate on

**Table 1** Function parameters description

| Parameter | Type | Description |
|---|---|---|
| mask | Boolean array | Control bits, specifying which elements should be processed |
| dst | Array | The output of the operation, storing the processed data |
| src | Array | The array containing the data to be processed |
| repeat_times | Integer | Specifies the number of times the operation is to be repeated |
| dst_rep_stride | Integer | The stride between each repeated operation in the destination array, set at a granularity of 32 bytes |
| src_rep_stride | Integer | The stride between each repeated operation in the source array, set at a granularity of 32 bytes |

the guidance provided by the measurement results procured from AIBench for real-world applications (Sect. 8).

## 4 Benchmarking compute units of AI processors

In this section, we measure the throughput of the main computing units in the AI core and summarize the impact of corresponding computing modes on time consumption during the testing process.

- Vector unit: The vector unit performs various mathematical computations, such as addition, subtraction, multiplication, division, and mathematical functions in a SIMD instruction manner. This unit is mainly used by AI operators such as ReLU, negation, and natural logarithm. The supported formats include unary, binary, scalar binary, scalar ternary, and reduction instructions. According to the documentation of Ascend 910, each cycle can complete the addition/multiplication of two 128-length float16 type vectors or 64 float32/int32 type vectors.
- Matrix unit: The matrix unit efficiently performs matrix operations. According to the documentation of Ascend 910, this unit can complete a $16 \times 16$ matrix multiplication with float16 inputs in one clock cycle. If the inputs are int8, the unit can complete a $16 \times 32$ or a $32 \times 16$ matrix multiplication in one clock cycle.

### 4.1 Vector unit

In this subsection, we validate the efficiency of vector unit, by measuring the pipeline throughput. In order to explore its full utilization, we use continuous addresses for both source data and destination data when running in a single AI core, in terms of the number of computations executed per cycle. We also measure the peak throughput in the vector computation unit with a full mask and 128 repeated vector iterations. In vector operation instructions, the elements participating in the computation are denoted by the mask, and the number of iterations is regulated by the repeat time. During the test, the number of elements involved in the computation is controlled by modifying the mask and repeat parameters, and the bandwidth changes are counted to obtain the computational bit width.

To demonstrate the pipeline throughput of the vector unit, the basic compute size of vector unit is examined using computations of varying length, and the time consumption for basic computation is explored by observing changes in time consumption.Fig. 2 shows the time consumption of vector calculation with varying masks. With varying mask settings

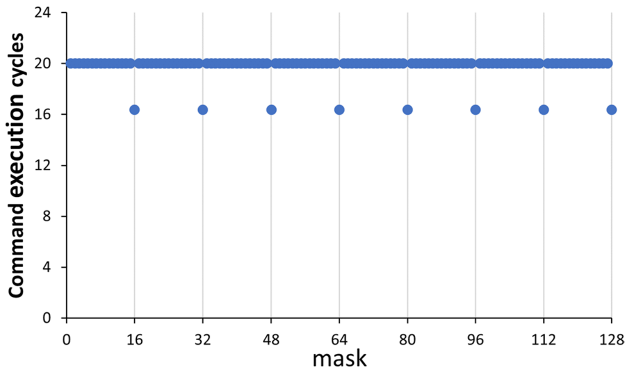
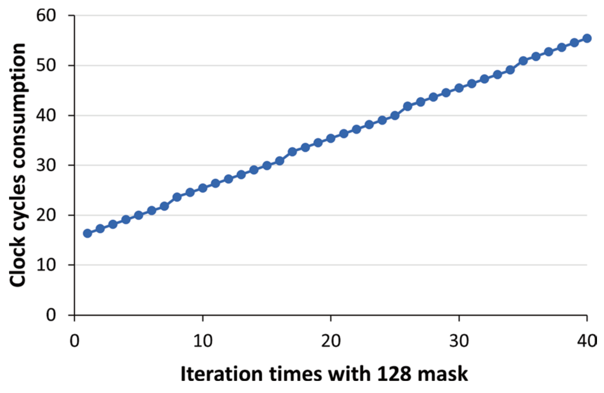
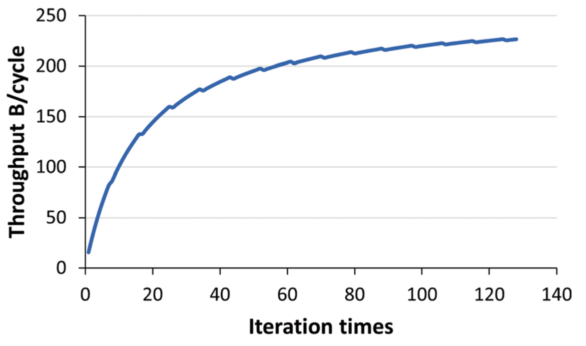


**Fig. 2** Time consumption of vector operation with varying mask



(a) Clock cycles consumption of vector calculation with varying iteration times



(b) Bandwidth of vector calculation with varying iteration times

**Fig. 3** Vector unit operations at different computational lengths

from 1 to 128, the time consumption will not decrease with the reduction of the amount of data involved in the calculation in a single iteration. Figure 3a, b respectively show the time consumption and bandwidth variation of vector computation as the computation length increases. Figure 3a illustrates that time consumption increases linearly by 256 B per cycle as the number of iterations changes. We make 2 major observations. First, vector unit data calculation parallelism is 256 B. As Figs. 2 and 3a show, only 256 B or larger workload variety will change clock cycle consumption. Second, the pipeline execution time for vector operations is one clock cycle, as illustrated in Fig. 3a. Considering the calculation parallelism and this pipeline execution time, we can conclude that the maximum pipeline throughput can attain 256 B per clock cycle. At a clock frequency of 900 MHz, the computing power of the vector unit is determined to reach 113 GFLOPS.

## 4.2 Matrix unit

In AI processors, the matrix computation unit is the principal source of processing power. Despite variations in the specific hardware circuitry across different AI processors, they are all capable of rapidly executing multiplication and addition operations on small matrices. Furthermore, these units accomplish matrix multiplication of arbitrary scale by segmenting the multiplication of large matrices into several smaller matrix operations. Consequently, to assess the matrix computation capabilities of AI processors, we should focus on the basic unit size and the number of clock cycles required for executing multiplication and addition operations on basic unit.

Matrix multiplication is a fundamental operation in linear algebra. It is a binary operation that produces a matrix from two matrices. The product of matrices A and B is denoted as AB. If A is an $M \times K$ matrix and B is a $K \times N$ matrix, the matrix product $C = AB$ is defined to be the $M \times N$ matrix. That is, the entry of the product is obtained by multiplying term-by-term the entries of the ith row of A and the jth column of B, and summing these K products. Therefore, the dimensions M, K, and N determine the shape of matrix multiplication.

In our experiments with the Ascend 910, due to limitations imposed by the instruction format, we selected 16 as the step size to vary M, K, and N, observing the execution time for matrix multiplication of different sizes. This approach enabled us to determine the base matrix operation size and the number of cycles consumed.

In this subsection, we deduced that within a single cycle, the matrix unit is capable of completing a $16 \times 16 \times 16$ matrix multiplication for float16 and $16 \times 32 \times 16$ multiplication for int 8. Furthermore, the computing power of a single AI core's matrix unit can achieve 3.6 TOPS for float16 and 7.2 TOPS for int8.

To get the pipeline throughput of the matrix unit, we investigate the basic unit size for matrix multiplication and throughput of the matrix unit. The experiment involved setting a fixed M dimension in matrix multiplication and transforming K and N dimensions in 16 steps. The relationship between K, N, and the number of multiplication in matrix

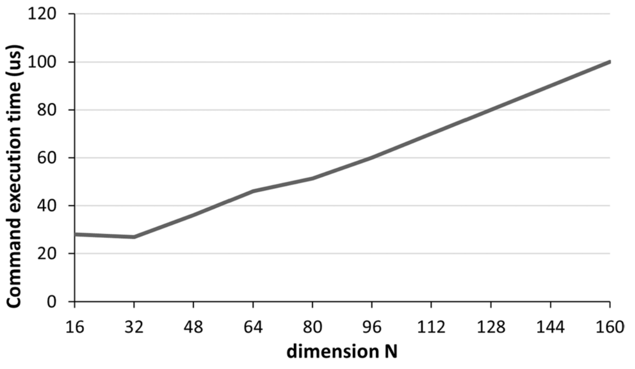


**Fig. 4** Time consumption of matmul operation with M = 160 and K = 144

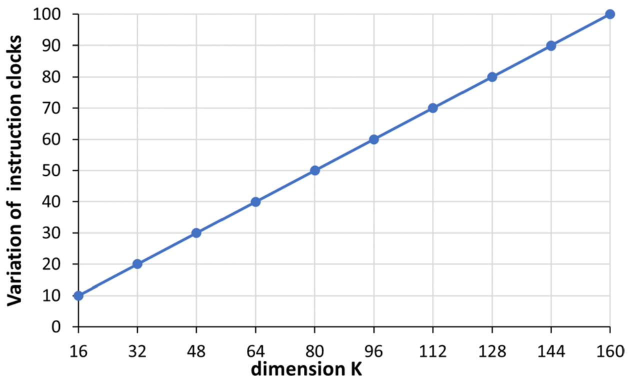


**Fig. 5** Variation in clock cycle consumption with the increase of N, given a set dimension M of 160

calculations is established by comparing time consumption with variations in N. The Fig. 4 illustrates that when M is fixed at 160 and N exceeds 112, there is a variation in time consumption corresponding to every increment of 16 in N. When M and K are fixed, an increase in N in matrix calculation will increase the corresponding matrix calculation amount. When M is fixed at 160 and K is set to 16, each increment of 16 in N results in an additional ten $16 \times 16 \times 16$ matrix operations. As can be seen from the Fig. 5, the time consumption of matrix multiplication escalates by 10 clock cycles. Additionally, when K assumes a larger value, the growth in clock cycles for each increment of N corresponds directly to the frequency of executing $16 \times 16 \times 16$ shaped matrix multiplication operations. We make a major observation that the matrix unit, while handling float16 data, can perform a $16 \times 16 \times 16$ matrix multiplication in a single clock cycle, with the resultant matrix values being accumulated within the same cycle. In the case of int8 data, a $16 \times 32 \times 16$ matrix multiplication can be completed in each clock cycle. Operating at a clock frequency of 900 MHz, this unit's computational power is capable of reaching 3.6 TOPS for float16 and 7.2 TOPS for int8.

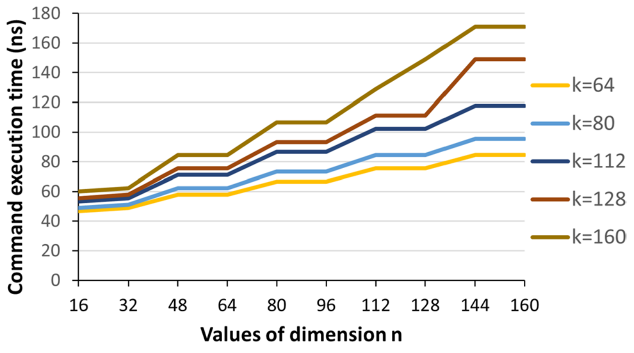


**Fig. 6** Time consumption of matmul operation with setting M = 16

During the testing process, we observe that the correlation between instruction execution time and variations in N is not entirely linear when M is small. As depicted in Fig. 6, there is a step-like increase in computation time as N increases when M = 16. We hypothesize that two 16 × 16 matrix multiplications can be completed simultaneously in the matrix computation unit. When M = 16 and N is an odd multiple of 16, the matrix computation unit cannot be fully utilized. Conversely, when M = 16 and N is an even multiple of 16, the matrix computation unit is fully utilized. Hence, when executing matrix multiplication with smaller sizes, we can optimize the performance of the matrix operation unit by ensuring an even number of matrix units are involved in the computation.

## 5 Benchmarking the on-chip memory subsystem of AI processors

In various AI processors, there are high-speed buffers around the computing units to adapt to the high throughput of computing units. These buffers cooperate with the data paths to provide input to the computing units and transmit output. The memory access characteristics and transmission bandwidth of these buffers have important guidance on how to choose the appropriate computing mode to fully exert the computing power of the computing units.

In AIBench, we count the time consumption changes of data handling at different granularities in the buffer area to analyze the buffer bit width and perform continuous data transmission between buffers to calculate the maximum bandwidth of each data path on the chip.

AI Core's internal buffer consists of the unified buffer, L1 buffer, and L0 out buffer. And AI core interacts with the bus through the BIU (bus interface unit) and achieves read-write operations with external storage units. In the following, we benchmark each component one by one.

### 5.1 Unified buffer

In this subsection, we determine the size of the unified buffer and investigate the bit width of the unified buffer.

By determining the maximum array that can be declared in the unified buffer, we get that the size of the unified buffer is 248 KB.

In this subsection, we investigate the bit width of the unified buffer. The experiment involves data transfer within the unified buffer with different block sizes. each block in the instruction has a size of 32 B. As depicted in Fig. 7, the time consumption varies with different burst size settings. We observe that instruction execution time increases in a step-wise manner, passing 16 blocks will increase one clock cycle. We calculate that the bit width of the unified buffer is 512 B.

### 5.2 L1 buffer

In this subsection, we get the size of the L1 buffer and examine the bit width of the L1 buffer.

By determining the maximum array that can be declared in the unified buffer, we get that the size of the L1 buffer is 1 MB.

We investigate the bit width of the L1 buffer. Given that TIK does not offer instructions for data transfer within L1 buffers, our experiment involves transferring data from the L1 buffer to the unified buffer while modifying the burst size. This method allows us to infer the bit width of the L1 buffer by determining the bandwidth between the L1 buffer and the unified buffer.

Figure 8a shows with the same number of transmissions, every four blocks take an additional cycle for data transmission from the unified buffer to the L1 buffer. We observe the throughput for data transmission from L1 buffer to the unified buffer is 128 B per clock cycle. Given that the unified buffer's measured bit width is 512 B, the bottleneck in

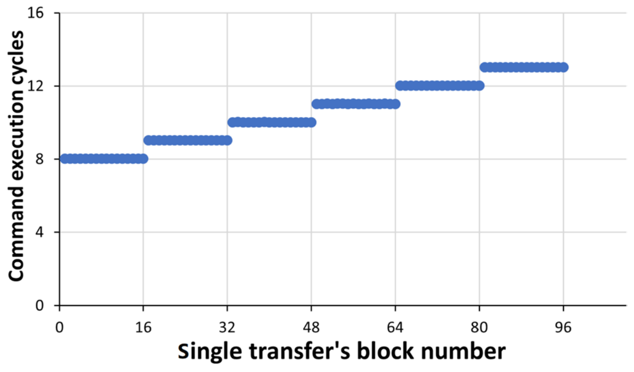
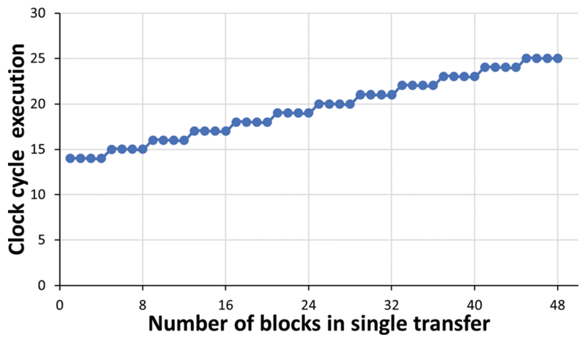
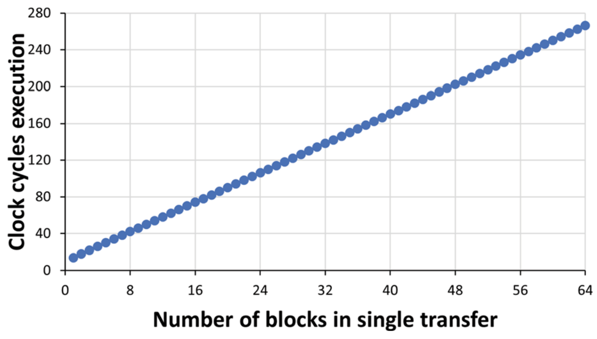


**Fig. 7** Clock cycle consumption of single transfers with different block sizes from unified buffer to unified buffer



(a) Clock cycle consumption from L1 buffer to unified buffer



(b) Clock cycle consumption from L0 out buffer to unified buffer

**Fig. 8** Clock cycle consumption for single transfers with different block sizes from L0 out and L1 buffer to unified buffer

data transmission from the L1 buffer to the unified buffer is attributed to the bit width of the L1 buffer. We conclude that the bit width of the L1 buffer is 128 B.

### 5.3 L0 out buffer

In this subsection, we get the size of the L0 out buffer and examine the bit width of L0 out buffer.

By determining the maximum array that can be declared in the unified buffer, we get that the size of L0 out buffer is 256 KB.

To get the bit width of L0 out buffer, the situation is similar to L1 buffer bit width getting, the bit width needs to be deduced from the bandwidth of the data transmission with L0 out buffer and unified buffer.

Figure 8b shows the time consumption for data transfer from the unified buffer to L0 out buffer. Transferring each 1024 B data takes 4.44 ns or four cycles. Therefore, the data bandwidth of the L0 out buffer should be 256 B per cycle. The storage mode of L0 out buffer is relatively unique. As a buffer for storing matrix calculation results, we observe that L0 out buffer only supports reads and writes the data in multiples of 1024 B size and requires four cycles for every 1024 B data.

### 5.4 On chip datapath

The data transmission within the AI core includes unified buffer and L1 buffer, unified buffer, and L0 out buffer. In the process of determining the bit widths of L0 buffer and L1 buffer, We have two observations. First, the pipeline throughput between unified buffer and L0 out buffer is 256 B per cycle. Second, the pipeline throughput between unified buffer and L0 out buffer is 128 B per cycle. We conclude that the maximum bandwidth between unified buffer and L0 out buffer reaches 230 GB/s, and the maximum bandwidth between unified buffer and L0 out buffer reaches 115 GB/s under 900 MHz clock frequency.

In addition, we test the bandwidth performance of the stride transfer mode by adjusting the spacing between the data blocks of the source and destination operands while keeping the number of transfer blocks and the length of each block constant. We observe that the transfer time is not affected by modifying the transfer spacing between data blocks as long as the source and destination operand positions satisfy the buffer alignment requirements (32 B alignment in unified buffer and L1 buffer, and 1024 B alignment in L0 out). We conclude that Within the inherent constraints of the TIK language, data transfer statements implemented can access all bank groups.

## 6 Benchmarking the global memory subsystem of AI processors

Global memory is the main storage part of the AI processor, which is located off-chip. Although it is slower than the buffer, it has a larger storage space. In AIBench, we calculate the bandwidth of data transfer in and out of the chip through continuous data reading and writing, then obtain the bandwidth of data transfer and the cache level in global memory by setting different sizes of working sets.

For Ascend accelerator, we conduct tests to investigate the memory access structure of the global memory. Concurrently, we measure the maximum transfer speed between the global memory and the bus under both single-core and multi-core scenarios.

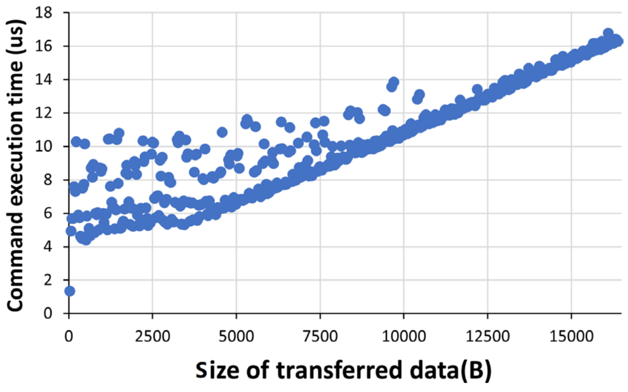
BIU manages the data interaction between the bus and the AI core. In this subsection, we benchmark the interaction between the AI Core and the bus to explore its maximum bandwidth for data read and write. We fix the number of data transfers and change the burst size in each transfer.

### 6.1 Single core data transmission

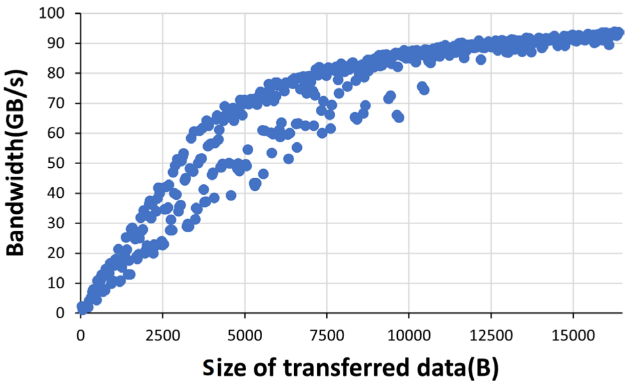
To examine the bandwidth from external memory to the on-chip buffer through BIU on the single core, we fix the times of data transfers and change the burst size for each transfer.

Figure 9a illustrates that using smaller blocks for data transfers can result in fluctuations in time consumption. However, as the volume of data escalates, we observe a linear increase in the time required for these transfers. Figure 9b shows the data transfer bandwidth varies with the length of each instruction. As the size of the data blocks increases, the transfer bandwidth gradually reaches its peak value. We have two observations. First, when transferring large amounts of data, the pipeline throughput can reach 108.7 GB/s, as indicated by the slope in Fig. 9a. Second, The bandwidth can reach 93.6 GB/s when transferring 16 KB of data in a single instrument as shown in Fig. 9b.

To assess the data transfer bandwidth between the internal buffer and external memory, we adopt a comparable methodology. Within the data path from global memory to the unified buffer, we observe a pipeline throughput of 103.7 GB/s. Moreover, for a single instruction that transfers 16 KB of data, the achievable bandwidth is 60.3 GB/s.



(a) Time consuming



(b) Bandwidth

**Fig. 9** Transferring data 100 times from external memory to AI core

## 6.2 Multi-core Data Transmission

To evaluate performance in a multi-core environment, we allocate the data transmission workload across all AI cores in the processor by distributing data blocks to each AI core. By monitoring how task duration fluctuates with changes in the number of distributed shards, we ascertain the maximum bandwidth achievable from the active cores to the bus.

To examine the access structure of the global memory, our experiment involves transferring an equivalent total amount of data from the global memory to the AI core, while varying the working set size within the global memory. During the computation, data is sequentially retrieved in 128 KB bursts from the designated range of the working set in the global memory and transferred to a fixed unified buffer address for data overwriting. In this experiment, attention is not centered on the specific division of labor among the cores; instead, the total time required for execution was measured.

Figure 10 illustrates that the bandwidth initially increases and then stabilizes as the working set size expands. When the working set size is less than 4 blocks (512 KB), the data transfer bandwidth exhibits an upward trend. Once the working set size surpasses 4 blocks, the data transfer bandwidth plateaus at 2850 GB/s. We make two observations from this experiment. First, when the working set size is below 512 KB, the data transfer bandwidth from the global memory to the AI core might be reduced due to reading conflicts in the global memory. As the size of the working set increases, so does the data transfer bandwidth. The maximum bandwidth from global memory to the bus is reached when the working set size exceeds 512 KB. Second, When the working set size exceeds 512 KB, the bandwidth stabilizes at its maximum value. We conclude that this maximum bandwidth can reach up to 2850 GB/s. Furthermore, within a dataset

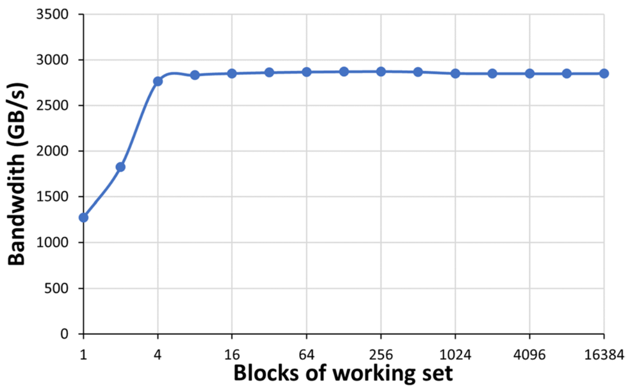
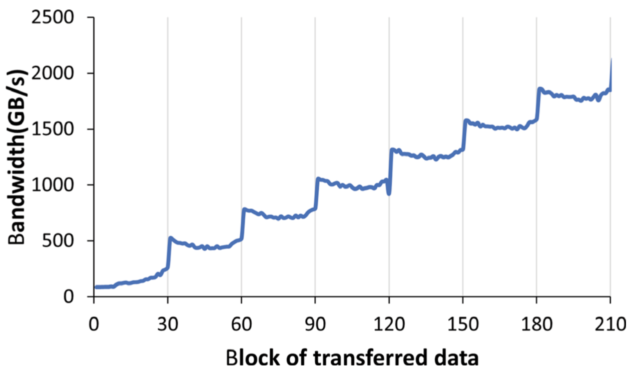


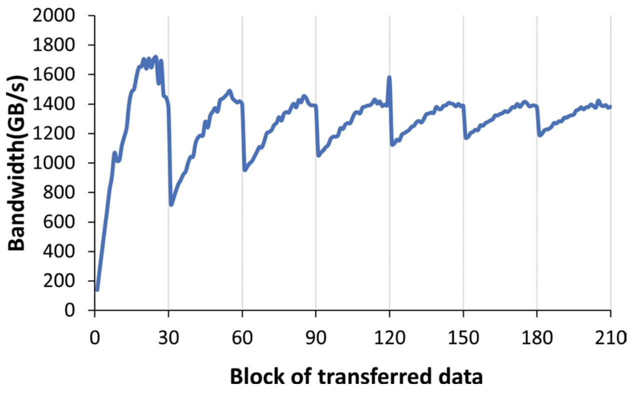
**Fig. 10** Under the multi-core configuration, the data bandwidth between global memory and L1 buffer as the number of 128 KB-sized data blocks varied in the working set

spanning 16,384 blocks (equivalent to 2 GB), the cache of the global memory seems to have no significant impact.

To determine the number of AI cores that can be actively engaged in computation, we assign multiple tasks to transmit data slices of 128 KB each to an AI core with several cores enabled. Figure 11 illustrates data transfer latency from external memory to an internal buffer. We observe that the instruction latency escalates in a step-wise fashion with a step size of 30 as the number of processing blocks augments. Our observations indicate that an equal distribution of data blocks among AI cores occurs when the total number of data segments is a multiple of 30. This uniform distribution is due to the system's architecture, where introducing an additional segment beyond a multiple of 30 necessitates engaging another AI core, leading to a noticeable increase in latency. Consequently, we conclude that the Ascend 910 chip utilizes 30 AI cores for computational tasks.



(a) Time consuming with 128KB block size



(b) Bandwidth with 128KB block size

**Fig. 11** Transferring data 100 times from external memory to AI core in multicore scenarios

## 7 Performance benchmarking of workloads with optimization strategies

Upon acquiring the performance metrics of computational units and transmission paths, we can evaluate various computationally demanding task flows. This allows us to ascertain the utilization and pipeline efficiency of the computational units and their associated transmission paths. This section evaluates the performance of AI processors under matrix and vector-intensive computational loads and tests the efficacy of two optimization strategies: double buffering and multi-core processing.

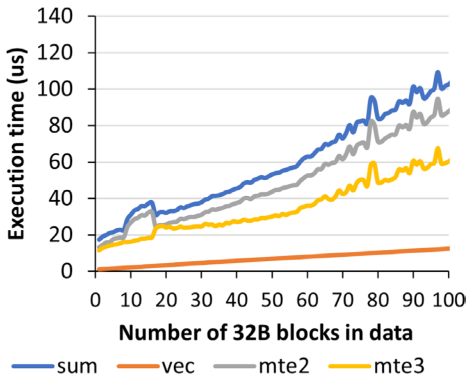
In the evaluation of computational load, data from memory is transferred to the on-chip for computation, subsequently, the result is relayed to the global memory outside the chip. The time consumed by each instruction is recorded as the computational load escalates.

### 7.1 Vector computational loads

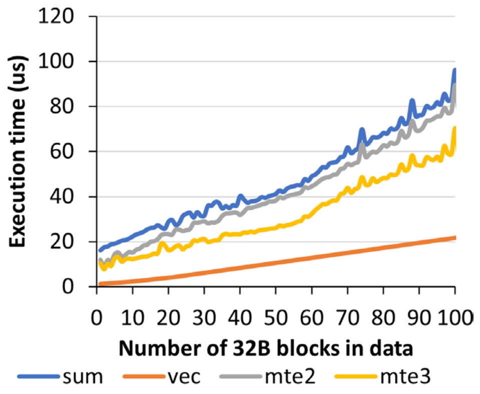
The vector computing unit, incorporated within AI processors, may offer less computational power compared to matrix computing units, yet it is capable of executing more flexible computations. In this section, we assess the performance of vector operation units under vector operation workloads. We evaluated the time efficiency of vector computing units during single-core execution and when utilizing double buffer in both single-core and multi-core environments.

Double buffer is a method for optimizing the performance of vector operations. A complete vector operation encompasses both data transfer and computation. Within the Ascend Accelerator employing double buffer, MTE2 is responsible for transferring data from external memory to the unified buffer, while the vector unit completes the computation and writes the results back to the unified buffer. Subsequently, MTE3 transfers the computed results back to external memory. The double buffer mechanism segments the unified buffer into two parts, enabling parallel execution of computation in one buffer and data transfer in the other, thereby diminishing the overall operation time.

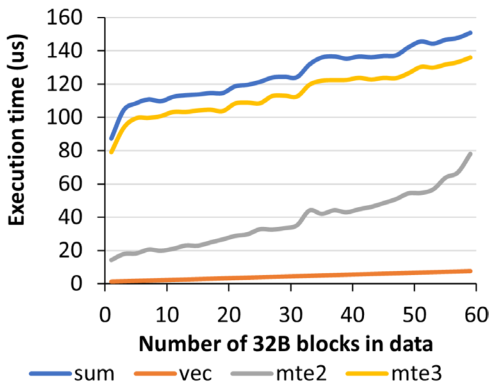
To assess the performance of AI processors in executing dense vector computations, we initially explored how time efficiency for vector computations of various lengths varies under a single kernel. As illustrated in Fig. 12a, performing vector computations without double buffering results in time consumption increasing linearly with the extension of computational length. The total operation time is approximately the sum of MTE2 time and computation time. Conversely, as depicted in Fig. 12b, with the implementation of double buffering, the total time



(a) Time consuming of once vector calculation without double buffer



(b) Time consuming of once vector calculation with double buffer



(c) Time consuming of once vector calculation with 30 AI cores and double buffer

**Fig. 12** Performance under vector computing load

efficiency is significantly reduced to nearly the duration of MTE2 time alone. We conclude that employing double buffering effectively overlaps computation time with data transmission.

To assess multi-core vector operations, a consistent load is applied to each AI core same as single-core testing. The average time taken for each instruction type in each core, along with the time needed for all cores to complete the entire task, was recorded. In the Ascend 910 chip, prior testing identified 30 operational cores. Thus, during multi-core testing, a workload equal to 30 times that of the single-core test was evenly distributed across these 30 AI cores.

We make two major observations in Fig. 13. First, implementing double buffering does not directly shorten the execution time for any specific type of instruction. Rather, it enhances overall efficiency by integrating data transformation and computation into a pipeline, thereby reducing the cumulative time needed for processes. Second, in executions involving multi-core processing of vector operations, the time taken for computation is similar to the duration of executing a computation instruction on a single core. However, during data transmission, due to issues such as bus contention, the time required can exceed twice that of a single-core computation, with data transmission emerging as the primary bottleneck. We conclude that the efficiency of vector computing units could be further enhanced by increasing the density of vector computations.

Given that the time expended on data input and output surpasses the computational time, we hypothesize that the utilization of a double buffer could potentially allow for the overlap of more computational operations with input and output, thereby enhancing pipeline efficiency.

We investigate the optimal pairing scheme for double buffering. The experimental setup involves varying the frequency of computations per data transfer into and out of the chip. As illustrated in Fig. 14, conducting 3 or 6 vector calculations in a single in-chip process does not lead to a bottleneck in the pipeline, and the overall time spent by the AI core remains constant. When the process involves nine vector computations, the time for vector

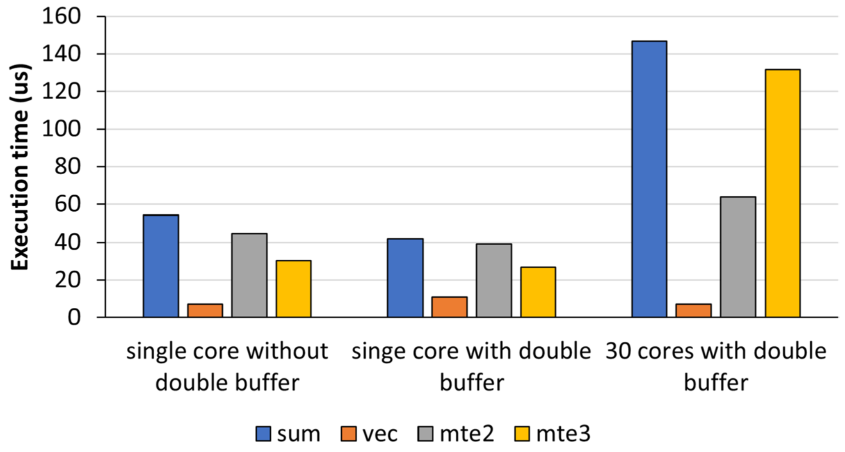
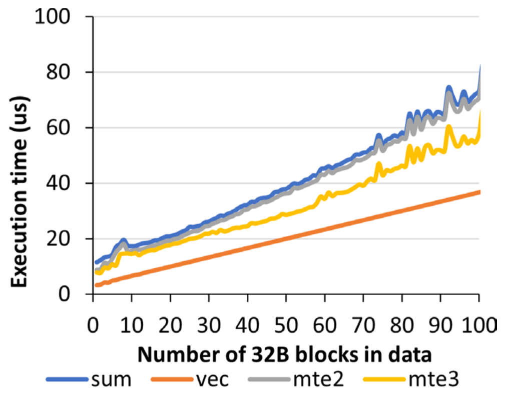
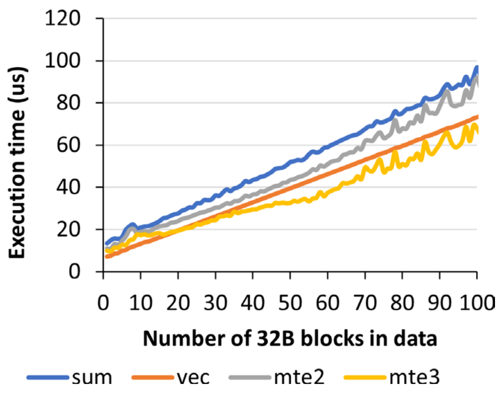


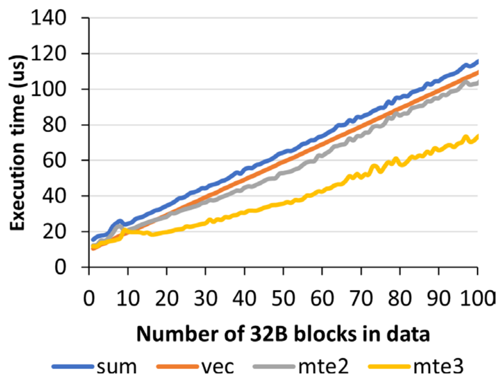
**Fig. 13** Execution time of each instruction under various optimization strategies



(a) Time consuming of three vector calculations with double buffer



(b) Time consuming of six vector calculations with double buffer



(c) Time consuming of nine vector calculations with double buffer

**Fig. 14** Performance with varying computational intensity under vector computational loads

computation surpasses the MTE2 time, thereby becoming the pipeline bottleneck and augmenting the total AI core time. Consequently, in the mode of slice input, vector computation, and slice output, feeding data for six vector computations can adequately fill the double buffer pipeline, thereby attaining peak computational efficiency.

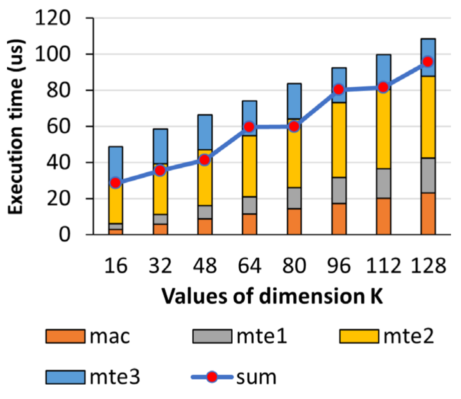
### 7.2 Matrix computational loads

Matrix computing units within AI processors are capable of delivering superior computational power. In this section, we assess the performance of these matrix operation units in managing matrix workloads in both single-core and multi-core configurations. Furthermore, we compare the parallel execution of instructions at varying computational densities.

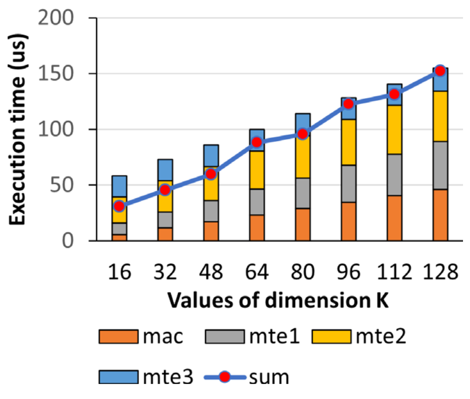
In the evaluation of matrix operations, we compute the length of the data stream and the number of data sharing operations via transformation. Subsequently, we assess the processor's performance under varying vector loads using both single-core and multi-core approaches.

For the Ascend 910 processor, we select matrix multiplication dimensions of 160 × k and k × 16 for our experiments. Our testing encompasses different scenarios: employing a single kernel to perform one matrix multiplication, using a single kernel for two matrix multiplications simultaneously, and deploying multiple kernels, each performing a single matrix multiplication.

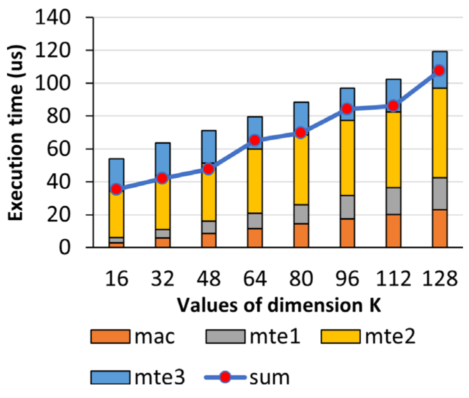
As depicted in Fig. 15, the total time efficiency increases in a stepwise manner as the value of k grows. We have three major observations. First, the dimensions of matrices involved in computations can significantly affect their performance, due to the presence of dual buffers that link the on-chip buffers with matrix computation units. Second, upon augmenting the density of matrix operation data sharding, the total time efficiency is amplified by an equivalent amount as a singular time efficiency. The transmission of accessible data is unable to mask the computational time. Third, When employing 30 cores and the average load per core aligns with that of a single-core test, it can attain a maximum data throughput of 608 GB/s.We conclude that choosing the appropriate mode for computation transmission is essential to fully harness the computational capabilities of AI accelerators.



(a) Time consuming of one matrix calculation



(b) Time consuming of two matrix calculations



(c) Time consuming of one matrix calculation with 30 cores

**Fig. 15** Performance with varying computational length under matrix computational loads

## 8 Optimization guidance for practical applications

This section will illustrate the optimization of fast Fourier transform (FFT) based on vector operation units, aiming to clarify the impact of AIBench measurement results on the optimization process.

The butterfly operation plays a pivotal role in the FFT algorithm, effectively simplifying the computation of the discrete Fourier transform (DFT). It achieves this by dividing an N-point DFT task into two sub-tasks, each dealing with N/2 points. This division process, characteristic of the butterfly operation, requires log(N) steps to complete for the entire dataset. In practice, calculating the operations for the xth layer involves processing $2^x$ elements, with each element undergoing both a multiplication and an addition to produce the output. This repetitive application of the same mathematical operations across the signal samples makes it ideally suited for vector processing units.

Experiments discussed in Sect. 8 reveal that performing over six vector operations inside the AI core allows for the overlapping of operation times of the vector unit with the data transfer times between the chip and global memory. In the FFT implementation employing vector operation units, the direct method involves executing a butterfly operation on each element after it enters into the AI core. Each element undergoes computation only twice during its transmission into and out of the chip. Under these conditions, data transmission emerges as the bottleneck, hindering the full exploitation of the computational capabilities of vector operation units.

To optimize the efficiency of vector operation units, it is necessary to execute over six vector operations on incoming data. This involves the data undergoing three stages of butterfly operations each time it enters the AI chip. Additionally, to comply with the prerequisites of butterfly operations, the input data must be a multiple of 8. As a result, the shape of the computational load is determined. Theoretically, this approach can increase the utilization rate of vector operation units by 60% compared to executing operations layer by layer.

The overarching similarity in the architecture of various AI processors enables the use of similar optimization techniques to boost calculation efficiency. In scenarios involving more complex computational loads, such as the need to execute format conversions within the AI core, employing on-chip buffers and multiple pathways within the core becomes essential. Under these circumstances, leveraging data from AIBench's benchmarks can guide the planning of the most efficient on-chip data transfer strategies.

## 9 Related work

To our knowledge, AIBench is the first benchmark testing tool for AI processor's underlying performance characteristics. Our work intersects with other work in the following areas (1) AI accelerator hardware architecture, (2) benchmarking neural network training and inference on AI accelerators, (3) hardware benchmarking on GPUs and FPGAs.

*AI accelerators.* Several AI accelerators (Liao et al. 2021; Jouppi et al. 2017; Norrie et al. 2021; Jouppi et al. 2023; Liao et al. 2019) have emerged in recent years. To adapt neural networks effectively, AI accelerators offer robust matrix computation capabilities. Despite variations in the specifics of their implementation, these accelerators share a similar foundational structure.

ıNeural network training and inference on AI accelerators. Previous works (Wang et al. 2020b; Reuther et al. 2019; Kumar et al. 2019; Sengupta et al. 2020; Zhang et al. 2020; Lu et al. 2022) have conducted end-to-end measurements of several AI accelerators for training and inference of neural networks. These works conduct a thorough comparison of various software libraries alongside an analysis of processor performance and efficiency across different deep-learning frameworks provided by multiple vendors. Specifically, the work (Lu et al. 2022) includes testing hardware utilization at the level of individual operator invocations beyond performing end-to-end testing on Ascend. Contrasting with the focus of these existing studies, AIBench emphasizes direct interactions with the underlying hardware, deliberately bypassing the complexities of higher-level frameworks.

*Hardware benchmarking on GPUs and FPGA.* Benchmarking has been widely used to determine the hardware organization of various processor architectures. For GPUs (Mittal and Vetter 2014; Jia et al. 2018; Mei and Chu 2016; Wong et al. 2010) and FPGAs (Zohouri and Matsuoka 2019; Manev et al. 2019; Wang et al. 2020a; Huang et al. 2022), benchmarking extends beyond computing speed also to encompass their storage structures. In a similar vein, AIBench employs comprehensive methods to assess cache configurations alongside computational performance across various hardware platforms.

## 10 Conclusion

We present AIBench, a benchmarking tool designed to reveal the underlying details of an AI processor. By utilizing AIBench to generate test operators, we obtain relevant hardware information. AIBench focuses on three primary areas of benchmarking. Firstly, AIBench assesses computing units by measuring vector calculations' parallelism and throughput, along with the operational modes and time efficiency of matrix computing units. Secondly, AIBench evaluates the AI core's internal buffer by determining each buffer's bit width and the data transfer bandwidth between buffers, which is informed by their sizes. Thirdly, AIBench examines data interactions within and outside the chip, capturing the maximum bandwidth achievable in such scenarios, how bandwidth varies with different data transmission strategies, and the effectiveness of data transfers in a multi-core configuration. In conclusion, AIBench rigorously evaluates the performance under computationally intensive workloads. Following this, we explore how the insights derived from these test results can be effectively applied to real-world applications.

The computing power of other AI processors such as GPUs, TPUs, and Cambricon is mainly provided by matrix computation units and vector computation units. Although the implementation of these computational units may vary, there is a commonality in the data flow when performing intensive workloads. So the testing methods of AIBench can also be applied to other AI accelerators.

**Acknowledgement** The work is supported by the National Key R&D Program of China (NO.2022ZD0119301).

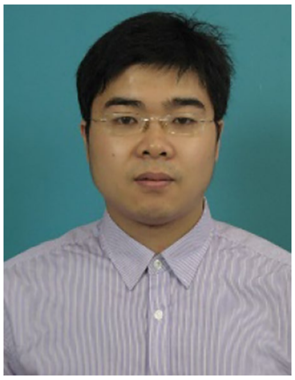
## Declarations

**Conflict of interest** Yang Xiao from Zhejiang University declares that there are no commercial or related interests representing conflicts of interest in the submitted work. Zeke Wang from Zhejiang University declares that there are no commercial or related interests representing conflicts of interest in the submitted work.

## References

Huang, H., Wang, Z., Zhang, J., He, Z., Wu, C., Xiao, J., Alonso, G.: Shuhai: a tool for benchmarking high bandwidth memory on fpgas. TC (2022)
Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D.P.: Dissecting the nvidia volta gpu architecture via microbenchmarking (2018). arXiv:1804.06826
Jouppi, N.P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., : In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 1–12 (2017)
Jouppi, N.P., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., et al.: Tpu v4: an optically reconfigurable supercomputer for machine learning with hardware support for embeddings (2023). arXiv:2304.01433
Kumar, S., Bitorff, V., Chen, D., Chou, C., Hechtman, B., Lee, H., Kumar, N., Mattson, P., Wang, S., Wang, T., et al.: Scale mlperf-0.6 models on google tpu-v3 pods (2019). arXiv:1909.09756
Liao, H., Tu, J., Xia, J., Zhou, X.: Davinci: a scalable architecture for neural network computing. In: Hot Chips Symposium, pp. 1–44 (2019)
Liao, H., Tu, J., Xia, J., Liu, H., Zhou, X., Yuan, H., Hu, Y.: 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), pp. 789–801 (2021). IEEE
Lu, W., Zhang, F., He, Y., Chen, Y., Zhai, J., Du, X.: Evaluation and optimization for huawei ascend neural network accelerator. Chin. J. Comput. **45**(8), 1618–37 (2022)
Manev, K., Vaishnav, A., Koch, D.: Unexpected diversity: quantitative memory analysis for zynq ultrascale+ systems. In: FPT (2019)
Mei, X., Chu, X.: Dissecting gpu memory hierarchy through microbenchmarking. IEEE Trans. Parallel Distrib. Syst. **28**(1), 72–86 (2016)
Mittal, S., Vetter, J.S.: A survey of methods for analyzing and improving gpu energy efficiency. ACM Comput. Surv. (CSUR) **47**(2), 1–23 (2014)
Norrie, T., Patil, N., Yoon, D.H., Kurian, G., Li, S., Laudon, J., Young, C., Jouppi, N., Patterson, D.: The design process for google's training chips: Tpuv2 and tpuv3. IEEE Micro **41**(2), 56–63 (2021)
Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., Kepner, J.: Survey and benchmarking of machine learning accelerators. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–9 (2019). https://doi.org/10.1109/HPEC.2019.8916327
Sengupta, J., Kubendran, R., Neftci, E., Andreou, A.: High-speed, real-time, spike-based object tracking and path prediction on google edge tpu. In: 2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), pp. 134–135 (2020). https://doi.org/10.1109/AICAS48895.2020.9073867
Wang, Z., Huang, H., Zhang, J., Alonso, G.: Shuhai: benchmarking high bandwidth memory on fpgas. In: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 111–119. IEEE (2020a)
Wang, Y., Wang, Q., Shi, S., He, X., Tang, Z., Zhao, K., Chu, X.: Benchmarking the performance and energy efficiency of ai accelerators for ai training. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pp. 744–751 (2020b). https://doi.org/10.1109/CCGrid49817.2020.00-15
Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying gpu microarchitecture through microbench marking. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), pp. 235–246. IEEE (2010)
Zhang, C., Zhang, F., Guo, X., He, B., Zhang, X., Du, X.: imlbench: a machine learning benchmark suite for cpu-gpu integrated architectures. IEEE Trans. Parallel Distrib. Syst. **32**(7), 1740–1752 (2020)
Zohouri, H.R., Matsuoka, S.: The memory controller wall: benchmarking the intel fpga sdk for opencl memory interface. In: H2RC (2019)

**Yang Xiao** received the bachelor's degree from Shandong University, Weihai, China, in 2021, where he is currently pursuing the Ph.D. degree. His current research direction is Approximate Nearest Neighbor Search (ANNS) in vectors.



**Zeke Wang** received the Ph.D. degree from Zhejiang University, Hangzhou, China, in 2011. He is a Research Professor at the Collaborative Innovation Center of Artificial Intelligence, Department of Computer Science, Zhejiang University, China. His current research interests mainly focus on building machine learning systems using heterogeneous devices, e.g., SmartNIC and SmartSwitch.