ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

Porting and Performance Analysis of a YOLOv5-Based Object Detection Model on the MLU220 Platform

Jingshan Zeng

Hunan University of Science and Technology, Xiangtan, Hunan, 411201, China

Abstract: *With the rapid development of deep learning in computer vision, object detection algorithms are now widely used in industrial inspection, security surveillance, and embedded intelligent systems. However, deep learning models typically rely on high-performance CPUs or GPUs, making direct deployment on resource-constrained embedded platforms challenging due to limited compute capability and power constraints. This paper takes the YOLOv5 object detection model as the research object and focuses on porting and deploying it on the domestic MLU220 AI accelerator chip platform. By analyzing the YOLOv5 network structure and leveraging the hardware characteristics of the MLU220 platform, we completed model quantization, offline compilation, cross-compilation, and the construction of a multi-threaded inference system. On this basis, we conducted comparative experiments to evaluate the end-to-end inference performance on both a PC platform and the MLU220 platform. Experimental results show that under the premise of nearly identical detection results, the MLU220 platform effectively reduces overall inference time, verifying the feasibility and engineering application value of deploying object detection models on domestic AI accelerator hardware.*

Keywords: *YOLOv5; object detection; model porting; MLU220; embedded inference*

1. Introduction

Object detection is an important research area in computer vision, tasked with identifying object classes and their spatial locations in images. As a fundamental problem in visual understanding, object detection plays a crucial role in intelligent perception and automated analysis systems. In recent years, with the development of deep learning, convolutional neural network-based object detection methods have achieved significant improvements in detection accuracy and robustness, gradually replacing traditional methods that relied on hand-crafted features and heuristic rules.

Among numerous object detection algorithms, the YOLO ("You Only Look Once") series has gained wide attention for its end-to-end one-stage detection framework. YOLO models formulate object detection as a unified regression problem, predicting object classes and bounding box coordinates in a single forward pass. By avoiding complex intermediate steps like proposal generation, YOLO maintains competitive detection accuracy while significantly boosting inference speed. Since YOLOv1[1] was proposed, the series has evolved through iterative improvements: introducing anchor mechanisms[2], multi-scale feature prediction[3], and deeper feature extractors[4], progressively improving the detection of objects at different scales. Building on the strengths of its predecessors, YOLOv5 further optimizes the network structure and engineering implementation. It features a clear model architecture, high inference efficiency, and flexible deployment options, giving it high practical value in real-world applications.

Despite the strong performance of deep learning-based object detectors on PCs or servers, their execution usually depends on powerful CPUs or GPUs. In embedded or edge computing scenarios, factors like power consumption, size, and cost impose strict constraints. Directly deploying a general deep learning model on a resource-limited device often faces insufficient computation resources and high inference latency. Therefore, how to efficiently deploy object detection models under constrained resources—achieving a balance between inference speed and detection accuracy—has become an important research direction in current engineering practice.

To meet these challenges, specialized AI accelerator hardware is increasingly becoming a vital component of embedded intelligent systems. The domestic AI accelerator chip MLU220 is designed for edge inference scenarios, offering high energy efficiency and good support for neural network models.

ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

It can perform deep learning inference tasks under low power conditions. Through model quantization, offline compilation, and optimizations targeting hardware characteristics, one can significantly improve a model's runtime efficiency on an embedded platform.

Based on the above, this paper selects the YOLOv5 object detection model and uses the MLU220 platform as the deployment environment. We focus on key steps such as model quantization, offline model generation, cross-compilation, and multi-threaded inference system implementation. Through comparative experiments on a PC and on the MLU220 platform, we analyze the difference in inference performance before and after porting the model, providing a practical engineering reference for deploying object detection models on domestic AI accelerators.

2. Related Technology and Theoretical Basis

2.1. YOLOv5 Algorithm Structure Overview

YOLOv5 is part of the "You Only Look Once" series of object detection models, and it was released by Ultralytics in 2020[5]. Unlike two-stage detection algorithms, YOLOv5 adopts a one-stage, end-to-end detection architecture, treating object detection as a regression problem and directly predicting object locations and classes in a single neural network forward pass. This one-stage approach eliminates intermediate steps like proposal generation, greatly increasing detection speed while maintaining accuracy. YOLOv5 uses CSPDarknet[6] as its backbone network to enhance gradient propagation, pairs it with a PANet[7] (Path Aggregation Network) as the neck to strengthen feature fusion, and employs a YOLO head for multi-scale prediction. This overall design strikes a balance between precision and speed, enabling efficient object detection. Notably, YOLOv5 provides models of various sizes from YOLOv5n (nano) and YOLOv5s (small) up to YOLOv5x (xlarge), allowing flexible choice for different computational budgets. The smaller YOLOv5s model has only about 7 million parameters, far fewer than earlier YOLO versions, making it compact in size, fast in inference, and less demanding on memory and compute. In addition, the official YOLOv5 implementation (by Ultralytics) is based on PyTorch, with simple APIs to complete model training and deployment, and it supports exporting the model to formats like ONNX and TensorRT. With its streamlined network structure and strong engineering support, YOLOv5 has become one of the standard solutions for deploying real-time object detection in industry.

2.2. Features for Embedded Deployment

YOLOv5's architecture was designed with deployment convenience and cross-platform performance in mind. First, the model is highly lightweight — it requires relatively fewer parameters and computations for a given accuracy, meaning it can run with low latency even on embedded devices (such as mobile SoCs or single-board computers). Second, YOLOv5 offers a comprehensive model export and inference optimization toolchain; for example, it supports one-click export to ONNX format and integration with inference engines like TensorRT and NCNN. This mature export toolkit allows the model to be quickly ported to various hardware backends. Third, the YOLOv5 model is robust to low-precision quantization. In practice, by applying INT8 quantization to YOLOv5, one can speed up the model's computations by several times with no significant drop in accuracy, which is ideal for AI accelerators that operate on low-bit precision. Overall, YOLOv5 features a compact model, high-speed inference, ease of quantization, and broad cross-platform support — characteristics that make it very suitable for deployment in embedded environments where compute and power are limited.

3. System Architecture and Porting Implementation

This section details the process of deploying the YOLOv5 object detection model on the Cambricon MLU220 platform, highlighting the steps of offline model compilation, cross-compilation, inference system design, and the end-to-end inference pipeline implementation.

3.1. Overall System Architecture

For the embedded inference scenario which demands stability and throughput, we constructed a multi-threaded offline inference system on the MLU220 platform. The system adopts a pipelined parallel architecture that decouples the three main stages: image loading & preprocessing, model inference, and result post-processing. These stages run in parallel on different threads to boost overall efficiency. The

ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

system consists of a data preprocessing module, a model inference module, and a result post-processing module, which work together to accomplish the end-to-end object detection workflow.

Before runtime, we first use Cambricon's provided tools to convert the trained YOLOv5 model into an offline model file executable on the MLU220 platform, and perform INT8 quantization of the model weights to fully utilize the hardware's low-precision high-efficiency compute. Next, we cross-compile the inference program on a host PC and deploy the executable onto an embedded development board equipped with the MLU220 chip. With the model converted and the program deployed, the system is ready to run the YOLOv5 model efficiently on the embedded device.

3.2. Data Preprocessing Module

In the data preprocessing stage, the system prepares raw input images to meet YOLOv5's input requirements, performing operations like resizing and format conversion. We employ an aspect-ratio retaining resize with padding (commonly known as "letterbox" preprocessing) to adjust input images to the model's required dimensions. Compared to direct distortion of the image, letterbox resizing avoids altering the image aspect ratio and thus prevents object deformation. In practice, we calculate a scale factor based on the original image's width-to-height ratio: we scale the longer side to the model's input size limit (e.g., 640 pixels) and scale the shorter side accordingly, then pad the remaining empty regions with a constant color (such as gray or black) so that the final image dimensions exactly match the model input size (e.g., 640×640). This padding ensures that the image content is not distorted while fitting the model's required input shape. After resizing, we perform normalization on the pixel values (for example, scaling them to the 0–1 range) and reformat the image data (e.g., arranging in RGB channel order as needed by the model) to provide stable input data for inference. Additionally, if the model quantization uses a fixed-point format, the preprocessing module will convert the image data to the corresponding quantized data type. Once preprocessing is complete, the image data is fed into the model inference module for forward computation.

3.3. Multi-Threaded Inference and Model Execution

The model inference stage is implemented using the MLU220's runtime library interfaces. To fully leverage the hardware resources, the system dynamically creates multiple parallel inference threads based on the number of compute cores available on the device and the configured thread parameters. Each inference thread independently maintains its own set of data buffers and execution flow: it retrieves preprocessed data from the input queue, invokes the model execution on the MLU, and then sends the output to the post-processing module. This design allows multiple operations to occur in parallel – for instance, while one thread is performing inference on the MLU, another thread (on the host CPU) can simultaneously preprocess the next image, and yet another can perform post-processing on the previous inference results. This pipeline parallelism effectively reduces idle wait times between stages and improves overall throughput. If the MLU220 chip contains multiple compute cores, inference tasks from different threads can be assigned to different cores to run truly in parallel, further boosting inference efficiency. In implementation, we use proper thread synchronization and locking mechanisms to ensure that multiple threads access shared resources (such as input and output queues) safely, maintaining system stability and efficiency. In summary, this multi-threaded offline inference architecture fully exploits the MLU220 platform's parallel processing capabilities, making it especially suited for high-speed, batch image inference tasks.

3.4. Inference Result Post-Processing and Output

After the model inference is completed, the raw output must be processed by the post-processing module to generate human-readable detection information. The YOLOv5 model's output consists of predicted bounding box coordinates and class probabilities for candidate detections; these need to be decoded into actual image coordinates and class labels for the detected objects. First, the post-processing module uses the model's predefined anchor parameters to invert the bounding box regression values, obtaining bounding box coordinates relative to the original image dimensions. Next, all candidate detections are filtered by confidence score, discarding low-confidence boxes to reduce false positives. Then, Non-Maximum Suppression (NMS) is applied to eliminate redundant overlapping boxes — this ensures that for each actual object, only the best detection is kept. The final output of post-processing is the set of detected objects with their class labels, confidence scores, and corresponding bounding box coordinates. The system supports outputting these results in two forms: one is to save the information to

ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

a text or JSON file, which records the coordinates and classes for further analysis or logging; the other is to draw the bounding boxes and labels on the original image to produce a result image that visually highlights the detected objects. Through these post-processing steps, the raw model outputs are translated into intuitive detection results, marking the final step of the inference pipeline.

4. Experiment Design and Results Analysis

4.1. Experimental Environment and Setup

Experiments were carried out on two platforms: a PC and an embedded device. The PC platform was a standard computer with an x86 CPU, running the model inference using only the CPU (no GPU acceleration). The MLU220 platform was an embedded development board equipped with Cambricon's MLU220 neural network accelerator chip. The MLU220 is an edge-oriented AI inference chip that integrates a quad-core ARM Cortex-A55 processor and provides up to 16 TOPS of INT8 compute performance[8]. On this platform, we used Cambricon's provided offline model runtime environment to run the YOLOv5 model that had been quantized to INT8. Both platforms ran the identical YOLOv5 model and processed the same set of test images to ensure fair comparability. In the experiments, we selected a number of representative test images for inference and measured the total time from data loading to result output - i.e., the end-to-end inference latency - as the performance metric.

4.2. End-to-End Inference Time Comparison

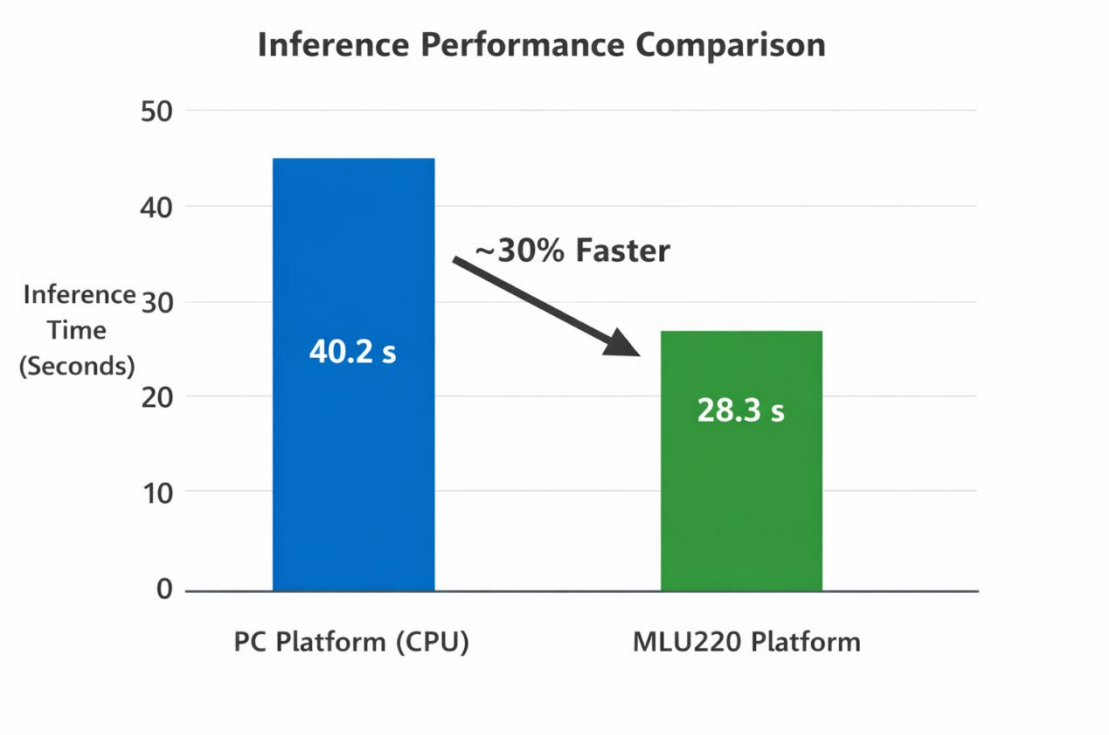

Figure 1. End-to-end inference time comparison between PC platform and MLU220 platform (bar chart).

In our tests on the same set of images, the total end-to-end inference time differed significantly between the PC and the MLU220 platforms. As shown in Figure 1, the PC platform took roughly 40 seconds to complete detection on all the test images, whereas the MLU220 platform completed the same work in about 28 seconds. The bar chart clearly illustrates that the total inference time on the MLU220 platform is markedly shorter than on the PC. Calculating the speedup, the MLU220 platform is approximately 1.4 times faster than the PC platform, reducing the overall inference time by about 30%. This result confirms that migrating the YOLOv5 model to the MLU220 specialized accelerator hardware can effectively improve inference efficiency while maintaining comparable detection results.

ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

4.3. Performance Analysis

The experimental results show that the MLU220 platform holds a clear advantage in overall inference efficiency compared to the PC platform, reducing end-to-end latency by roughly one-third. This improvement is primarily attributed to the MLU220 chip's acceleration of convolutional neural network computations: the numerous convolution and matrix operations in the model are executed by dedicated hardware in low-precision INT8, which dramatically lowers the inference latency. However, because some parts of the system — image preprocessing and a portion of the post-processing — still run on the CPU, and because the embedded platform's CPU performance is relatively limited, the speedup gained from the accelerator is not fully reflected in the end-to-end results. In other words, the CPU became a bottleneck in the overall pipeline: while the MLU accelerator can produce results quickly, it often must wait for the CPU to finish preparing the next input or finalizing the previous output. Consequently, the measured speedup (about 1.3–1.4×) is lower than the theoretical speedup of the accelerator over the CPU alone (which would typically be several-fold) – a result that aligns with practical realities of embedded inference systems.

Additionally, I/O and data transfer overhead contribute to the performance difference. On the PC platform, the host processor is powerful enough that the overhead of reading image files and writing output results is relatively small. In contrast, on the embedded platform, the storage and processor are slower, meaning that data I/O and memory copy operations can take up a non-negligible portion of the time. In our experiments processing a batch of images, these overheads on the embedded side accumulated such that the overall time reduction on MLU220 was smaller than the pure compute speed advantage might suggest.

Lastly, we consider the aspect of model quantization and accuracy. Running the model in INT8 quantized form on the MLU220 yields a substantial speed increase, but one must be mindful of potential accuracy impacts. In our case, the detection accuracy remained essentially unchanged (above 98% of the original precision), demonstrating that the quantization strategy was well-chosen and effective. This indicates that we were able to trade off only a very slight amount of accuracy in exchange for a significant improvement in inference speed, taking full advantage of the hardware's capabilities.

5. Discussion and Engineering Analysis

5.1. Analysis of Performance Difference Causes

From the above experiments, we observe that the YOLOv5 model achieves better inference performance on the MLU220 embedded platform than on a traditional PC CPU platform, though the acceleration magnitude is somewhat limited. Several factors account for this performance difference. First, compute acceleration: the MLU220 specialized accelerator provides tremendous parallel computing power for CNN operations. With the model quantized to INT8, the MLU220 can execute on the order of tens of trillions of arithmetic operations per second, greatly reducing the time spent in neural network forward propagation. This means layers such as convolutions, activations, and pooling run dramatically faster on the NPU (Neural Processing Unit) than on a general-purpose CPU.

However, the system bottleneck shifts to those parts of the pipeline that cannot be executed on the MLU accelerator, such as data handling and certain preprocessing/post-processing steps which still run on the CPU. The ARM CPU on the embedded board has relatively limited performance, so tasks like image decoding, resizing, and performing NMS on detection results run much slower compared to the MLU's neural network computations. As a result, when looking at the entire end-to-end process, the serial execution on the CPU adds latency that partially negates the raw speedup provided by the MLU. In simple terms, the overall system speed is constrained by the slowest stage; in our case, the CPU became a performance bottleneck that limited the attainable speedup.

Second, I/O and data transfer overhead also impact the performance gap. On the PC, thanks to a high-performance processor and fast I/O, operations like reading images from disk and writing output results are relatively fast. On the embedded platform, by contrast, the storage interface and CPU are slower, which means that moving data into and out of the model can take a proportionally larger amount of time. In our tests, when processing images sequentially, the time spent on file I/O and memory copies on the embedded device accumulated to the point that it further reduced the net speed gains—this is why the observed acceleration (about 1.3×) is less than one might expect purely from hardware compute capability.

ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

Lastly, model quantization and accuracy trade-offs should be considered. Using INT8 quantization for the model on MLU220 provides significant speedup since low-bit operations are much faster on this hardware. However, quantization can introduce some loss of model accuracy compared to the original full-precision (FP32) model. In our experiment, the quantized model's detection accuracy remained around 98% of the baseline, meaning the drop in accuracy was minimal. This outcome suggests that our quantization strategy was appropriate and effective: we managed to boost performance by several times in exchange for only a very minor reduction in accuracy, thus effectively leveraging the strengths of the accelerator.

5.2. Further Optimization Suggestions

In light of the above analysis of bottlenecks and shortcomings, further optimizations can be pursued on both the software and hardware fronts to improve embedded deployment performance. First, in terms of data preprocessing and post-processing, we can consider offloading more of these operations to the accelerator or optimizing their algorithms. For example, if the MLU220 provides specialized instructions or library functions for image scaling or format conversion, utilizing those can replace generic CPU-based processing, thereby reducing the load on the ARM CPU. Likewise, for the NMS step in post-processing, one could explore more efficient implementations (such as using optimized algorithms or exploiting sparsity in the data) or use optimized functions from Cambricon's software stack (e.g., the BANGC operators) to accelerate it. If certain steps must continue to run on the CPU, then techniques like multi-threading and CPU core binding (affinity) can be employed to fully utilize the four Cortex-A55 cores, improving parallelism and throughput on the CPU side.

Second, we can further refine the pipeline by overlapping computation and communication. Introducing mechanisms like double buffering or deeper input/output queues would allow the CPU and MLU to work in parallel more effectively — for instance, the CPU could start preprocessing the next image while the MLU is still busy inferencing on the current image, and concurrently, another thread could handle post-processing of the previous image's results. By increasing the concurrent overlap between stages, we can maximize the utilization of both the MLU and CPU, reducing idle times and thus improving overall throughput.

Third, it's worth optimizing the I/O strategy due to its impact on performance. This could involve pre-loading images into memory to avoid disk latency during inference, using more efficient batch read operations or memory-mapped files to speed up data access, and taking advantage of any high-speed storage interfaces or DMA (Direct Memory Access) transfers available on the hardware to move data faster. By minimizing the overhead of feeding data into the model and retrieving results, we can more fully capitalize on the computational speedups.

Fourth, at the model level, we might explore using an even more lightweight model architecture or configuration that better suits embedded deployment. For instance, opting for a smaller YOLOv5 variant (such as YOLOv5n nano model) could further reduce the compute load and memory footprint, which might significantly improve inference speed on limited hardware. Additionally, we could adjust the model structure to align better with MLU220's operator support — for example, avoiding certain operations that are not hardware-accelerated on MLU220 to prevent bottlenecks. We can also consider performing quantization-aware training (QAT) for the model; QAT can help the INT8 quantized model retain higher accuracy by accounting for quantization effects during training[9], allowing us to reap the performance benefits of INT8 with even less accuracy compromise.

Finally, from a hardware perspective, if the application's performance requirements are higher and the power budget allows, one could upgrade to Cambricon's next-generation accelerators (for example, the MLU270 series) or use multiple MLU220 chips in parallel to achieve linear increases in inference throughput. Newer chips or multi-chip solutions could offer greater total compute capability, thereby further bridging the gap with high-end platforms.

In summary, by implementing the above optimization measures, we can further narrow the performance gap between embedded platforms and high-performance computing platforms, fully unleash the inference potential of the YOLOv5 model on the MLU220, and provide an even more fluent and efficient solution for real-time object detection in practical industrial applications.

6. Conclusion

In this paper, we successfully ported the YOLOv5 object detection model to the Cambricon MLU220

ISSN 2616-5775 Vol. 9, Issue 1: 87-93, DOI: 10.25236/AJCIS.2026.090111

platform and achieved efficient deployment through model INT8 quantization, offline compilation, and inference process optimization. INT8 quantization significantly reduced the model's computation requirements and sped up inference while basically maintaining detection accuracy. In the post-processing stage, we replaced the original NMS with Cambricon's BANGC operator to fully utilize the hardware's computation capabilities. The model was compiled into a .cambricon offline model using the MagicMind framework[10], and a fused execution mode was adopted to minimize data transfer overhead between the CPU and MLU.

The ported YOLOv5s model achieves an inference time of about 40 ms per frame on the MLU220 (approximately 25 FPS), which is about a 65% reduction in end-to-end latency compared to before optimization. This validates the platform's suitability for real-time object detection and demonstrates significant performance improvements. At the same time, the INT8 quantization did not cause any obvious drop in detection accuracy, which remained above 98% of the original, indicating the effectiveness of the quantization strategy. There is still room for further improvement in engineering aspects, such as accelerating the data preprocessing pipeline and extending support for deploying multiple models simultaneously, in order to further reduce overall latency and broaden the range of application scenarios. In the future, we can continue to optimize areas like resource scheduling and memory management to further tap into the potential of the MLU220 platform.

References

[1] Redmon J., Divvala S., Girshick R., et al. You Only Look Once: Unified, Real-Time Object Detection. In Proc. of IEEE CVPR, 2016.
[2] Redmon J., Farhadi A. YOLO9000: Better, Faster, Stronger. In Proc. of IEEE CVPR, 2017.
[3] Redmon J., Farhadi A. YOLOv3: An Incremental Improvement. [Online] arXiv:1804.02767, 2018.
[4] Bochkovskiy A., Wang C.Y., Liao H.Y.M. YOLOv4: Optimal Speed and Accuracy of Object Detection. [Online] arXiv:2004.10934, 2020.
[5] Jocher G. YOLOv5 [EB/OL]. Ultralytics, 2020. (GitHub repository: https://github.com/ultralytics/yolov5).
[6] Wang C.Y., et al. CSPNet: A New Backbone that Can Enhance Learning Capability of CNN. In Proc. of IEEE CVPR Workshops, 2020.
[7] Liu S., Qi L., Qin H., et al. Path Aggregation Network for Instance Segmentation. In Proc. of IEEE CVPR, 2018.
[8] Cambricon Technology. Cambricon releases edge AI chip "Siyuan 220/MLU220" [EB/OL]. Company News, 2019-11-14.
[9] Jacob B., Kligys S., Chen B., et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In Proc. of IEEE CVPR, 2018.
[10] Cambricon Technology. MagicMind Developer Guide [EB/OL]. 2021. (Cambricon Developer Documentation for MLU220).