A Thread-Level Stream Scheduling Method for Accelerating LVMs' Inference on a Resource-Constrained Platform

YIJIE CHEN, Northwest A&F University, Yangling, China
JIAQI HAN, Northwest A&F University, Yangling, China
BIN LIU*, Northwest A&F University, Yangling, China
XINZHE ZHANG, Northwest A&F University, Yangling, China
ZIJIAN HU, Northwest A&F University, Yangling, China
RONGYU DOU, Northwest A&F University, Yangling, China
KEQIN LI, Department of Computer Science, State University of New York at New Paltz, New Paltz, United States

As a new generation of edge devices, the integrated CPU/GPU architecture has opened up new opportunities for deploying different scale vision models. In order to reduce models' inference time on the integrated devices, this article first compresses deep learning models using model quantization. The quantization process greatly reduces the computation requirements of a model, which enables its deployment on embedded development boards. However, quantization also leads to lower GPU resource utilization during inference on integrated devices. This insufficient utilization results in slower inference speed. To address this problem, this article first depicts the data flow of model inference within an integrated device. Secondly, this article implements a unified memory management between the CPU and GPU based on managed memory strategy. Finally, this article designs a thread-level stream scheduling method to improve GPU utilization and throughput during model inference in a pipeline way. Experimental results show that the proposed method achieves a 2x-10x improvement in throughput compared with the TensorRT's default scheduling method, which is crucial for realizing real-time inference tasks on edge devices.

CCS Concepts: • **Computer systems organization** → *Embedded systems*; • **Computing methodologies** → *Parallel computing methodologies;*

Additional Key Words and Phrases: Deep neural network, TensorRT, model inference, unified memory management, heterogeneous processing

Yijie Chen and Jiaqi Han contributed equally to this work.

This work was supported by the National Natural Science Foundation of China under the Grants No. 62376226, by the Natural Science Basic Research Program of Shaanxi under the Grants No. 2025JC-YBQN-888, by the Technology Innovation Leading Program of Shaanxi under the Grants No. 2024QCY-KXJ-094, and by the Xi'an Key Technology Research Projects for Key Agricultural Industry Chains under the Grants No. 2024JH-NYZD-0027.

Authors' Contact Information: Yijie Chen, Northwest A&F University, Yangling, Shaanxi, China; e-mail: yijiechen@nwafu.edu.cn; Jiaqi Han, Northwest A&F University, Yangling, Shaanxi, China; e-mail: hanjiaqi@nwafu.edu.cn; Bin Liu (corresponding author), Northwest A&F University, Yangling, Shaanxi, China; e-mail: liubin0929@nwsuaf.edu.cn; Xinzhe Zhang, Northwest A&F University, Yangling, Shaanxi, China; e-mail: zhangxz2113jg@nwafu.edu.cn; Zijian Hu, Northwest A&F University, Yangling, Shaanxi, China; e-mail: 2021056015@nwafu.edu.cn; Rongyu Dou, Northwest A&F University, Yangling, Shaanxi, China; e-mail: dourongyu@nwafu.edu.cn; Keqin Li, Department of Computer Science, State University of New York at New Paltz, New Paltz, New York, United States; e-mail: lik@newpaltz.edu.

ACM 1539-9087/2026/05-ART51
https://doi.org/10.1145/3771550

ACM Reference Format:
Yijie Chen, Jiaqi Han, Bin Liu, Xinzhe Zhang, Zijian Hu, Rongyu Dou, and Keqin Li. 2026. A Thread-Level Stream Scheduling Method for Accelerating LVMs' Inference on a Resource-Constrained Platform. *ACM Trans. Embedd. Comput. Syst.* 25, 3, Article 51 (May 2026), 27 pages. https://doi.org/10.1145/3771550

1 Introduction

As one of the most advanced techniques in the field of **artificial intelligence** (**AI**), deep learning has made remarkable achievements in numerous industries in recent years. Deep learning is the key engine driving technological progress in edge application areas such as automatic driving and **unmanned aerial vehicle systems** (**UAVs**), where **deep neural networks** (**DNNs**) have significantly improved the accuracy of vehicle perception and decision-making through in-depth learning of complex driving scenarios, injecting new vitality into the development of intelligent transportation systems [3, 8]. Similarly, object tracking technology and face recognition technology for UAVs have also achieved significant performance improvements thanks to the addition of DNN [12, 21].

Researchers have extensively applied DNNs to integrate AI into our daily lives. It should be mentioned that these studies are based on cloud computing [4, 25, 27] and edge computing methods in the field of **Internet of Things** (**IoT**) [9, 10, 20] to realize real-time inference of AI applications. However, real-time inference of the DNN models performed in the cloud incur excessive response latency, which critically depends on forward and backward communication between the data source and the cloud, while raising privacy issues for user sensitive data. Hence, researchers directly deploy deep learning models on edge devices to avoid the above problems. By collecting and processing data in real time through sensors, an edge device can be built as a fast and efficient deployment platform, especially for computer vision tasks that require hard real-time performance. However, edge devices are so resource constrained to prevent the deployment of computationally intensive DNNs, such as the vision models with different scale.

The emergence of integrated CPU/GPU architectures ([14–16, 19]) has opened up new opportunities for deploying the DNNs' model on edge devices with their advantages in terms of computation, memory, and power. Naturally, it is not feasible for an integrated platform to perform the computing task using the on-chip CPU or GPU independently. As shown in Figure 1, when we perform the inference tasks of ResNet50 and WideResNet on the Jetson Xavier NX, executing DNN workload on CPUs is two orders of magnitude slower than running it on GPUs. In order to achieve real-time inference of DNNs, utilizing the on-chip resources collaboratively is crucial. Besides, a deep learning framework is needed to schedule the on-chip computing resources for performing the inference task of DNNs. PyTorch, a heavy-weight deep learning framework designated for servers, generates models with default parameter data type of float32. Due to the large number of parameters and computational complexity of deep learning models, it is often necessary to quantize their parameters to meet the requirements for deployment on edge devices. However, PyTorch itself does not support the acceleration effect of quantized models on hardware.

In order to enjoy the acceleration potential of quantized models on NVIDIA GPUs, NVIDIA launched the TensorRT framework, which is a high-performance deep learning inference optimizer and runtime engine and provides APIs to express deep learning models via the ONNX parser that allows TensorRT to optimize and run them on an NVIDIA GPU. PyTorch is used as the front-end and TensorRT is used as the back-end to complete the deployment of the deep learning model. During the inference process, the data flow of the default process of the PyTorch model can roughly be divided into three steps. Firstly, PyTorch uses a library (Torchvision) to load input data and converts the data into Tensor format. Secondly, a DNN model is loaded into the memory of the GPU.

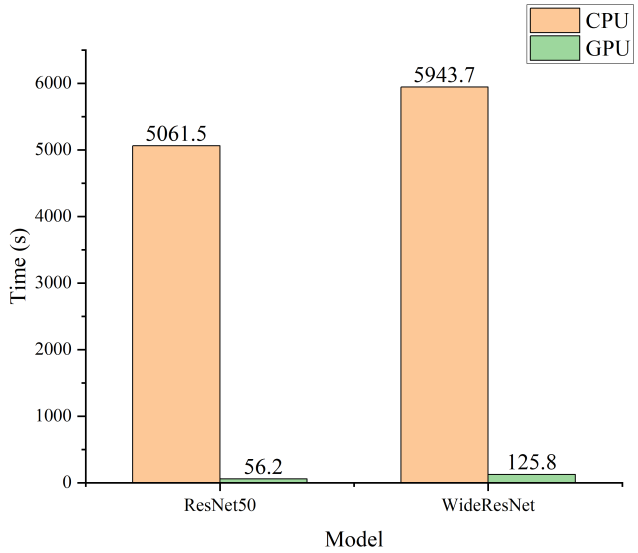


Fig. 1. The inference time for two DNN models using the CPU and GPU of the Jetson Xavier NX, respectively.

Thirdly, input data is transferred from the CPU memory to the GPU memory for GPU computation. TensorRT uses multi-threading to call a single CUDA stream for computing tasks by default, which simplifies concurrent inference management while maintaining efficient GPU utilization. However, all tasks (data transfer, memory operation, kernel tasks, etc.) in a single CUDA stream must be carried out strictly sequentially, preventing utilization of the GPU's asynchronous capabilities, causing suboptimal utilization of memory bandwidth (the PCIe bus sits idle during kernel execution, preventing concurrent data transfers for the next batch). Furthermore, when multiple threads submit tasks to the same stream, the CUDA runtime requires internal lock synchronization, leading to thread contention. This significantly degrades GPU computing efficiency, particularly on unified memory architectures. Therefore, the default scheduling method in TensorRT for model inference cannot efficiently utilize hardware resources on edge devices, leading to generally low utilization of GPU resources during model inference. For example, on a Jetson Xavier NX, using the PyTorch framework, the GPU utilization of three classic DNN models during the inference process on the CIFAR-100 dataset is depicted in Figure 2. It is obvious that the GPU utilization on the Jetson Xavier NX depends on the model architecture itself, and the current GPU resource usage remains suboptimal (For instance, for GoogleNet, the peak memory is only 60%). When the batch size (how many data samples are used per training iteration, dubbed as Bs) is set to 1, GPU utilization is typically not high, sometimes even below 50%. Although this situation can be slightly mitigated by increasing the batch size (as shown in Figure 2) or adopting a larger model (as shown in Figure 3(a)), the on-chip resources of the integrated device are still not fully used. Increasing the batch size or adopting a larger model increases the computational costs and memory costs borne by the edge platform.

To reduce the computational cost and memory cost for deploying a DNN model on an edge device, researchers have proposed multiple model compression methods such as pruning, quantization, low-rank decomposition, and knowledge distillation [5, 13, 28], etc. These methods optimize the inference process by reducing DNN models' resource consumption. However, for any deep learning model quantized by TensorRT, although the quantization process greatly reduces the amount of computation required for inference tasks and accelerates the inference process, it also leads to a decrease in GPU resource utilization. Since TensorRT's default scheduling mechanism uses only a

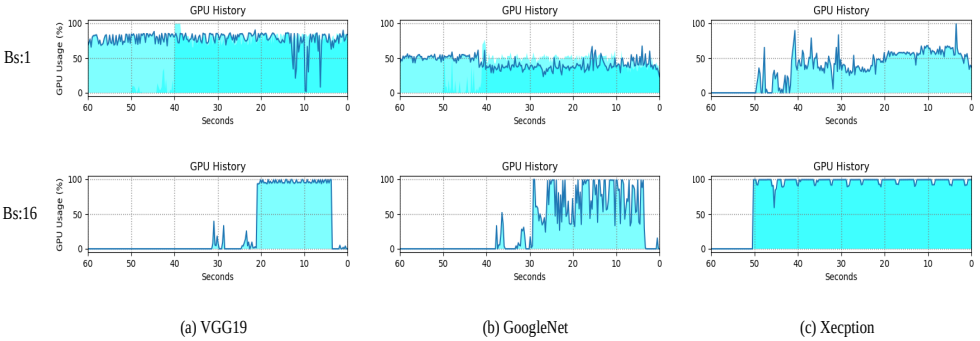


Fig. 2. The GPU utilization for inferencing DNN models on Jetson Xavier NX. These models are tested on the CIFAR-100 dataset. *Bs* represents the batch size, which is set as 1 and 16, respectively.

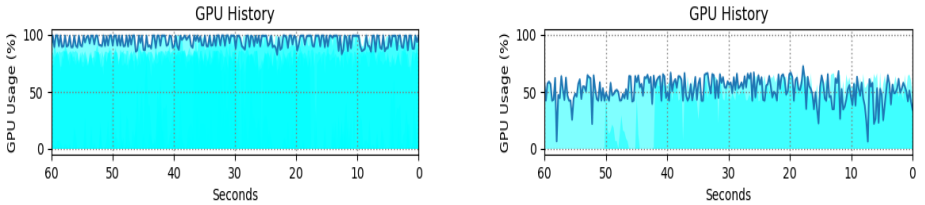


Fig. 3. The GPU utilization for deploying WideResNet with/without model compression on Jetson Xavier NX. WideResNet is tested on the CIFAR-100 dataset and the batch size is set as 1. The compression part applies INT8 quantization.

single CUDA stream during the process of model inference and all tasks in a single CUDA stream are addressed strictly sequentially, preventing utilization of the GPU's asynchronous capabilities, causing under utilized memory bandwidth, the GPU utilization remains low. As shown in Figure 3, the WideResNet model is compressed through INT8 quantization and optimized by the TensorRT's serialization engine. The GPU utilization of the compressed model is significantly reduced compared with the original WideResNet model during the inference process.

In order to improve the GPU utilization for the DNNs' real-time inference, this article proposes a novel thread-level CUDA stream scheduling method based on unified memory. Firstly, this article analyzes the general inference process of the TensorRT engine of a PyTorch model in the integrated computing platform. Considering, the problem of slowing down inference due to inter-thread communication in PyTorch can be mitigated by optimizing GPU memory management to reduce overhead [2]. Secondly, this article designs a heterogeneous collaborative computing method based on unified memory, aiming to reduce communication overhead. Finally, this article employs multiple threads with multiple CUDA streams to enhance GPU concurrency, which improves GPU utilization and makes full use of the computing resources of an integrated device. The contributions of this article can be summarized as follows:

— The inference data flow of TensorRT engine for a PyTorch model on an integrated device is analyzed to demonstrate how the computing tasks of model inference is scheduled on heterogeneous resources.

— Three different typical GPU memory management methods are analyzed, and the managed memory strategy is adopted to implement unified memory management between on-chip CPU and GPU.
— A heterogeneous pipeline-style stream scheduling method is designed to improve the utilization of on-chip resources, which adopts a thread-level stream scheduling algorithm to maximize the GPU throughput for various DNN models.

Extensive experiments are conducted on the Jetson Xavier NX platform to demonstrate the effectiveness of the proposed method. The remaining parts of this article are organized as follows. Section 2 demonstrates the related works. Section 3 describes the proposed method in details. Section 4 provides the experimental platform and the details of implementations, and presents relevant performance results from various performance metrics. Finally, Section 5 discusses the advantages of the proposed method, emphasizing how to extend it to allow dynamic runtime scheduling.

2 Related Work

Delia et al. proposed a co-execution strategy to boost the performance of model inference on Jetson TX2 device with heterogeneous processors [22]. They pipelined computational tasks of heterogeneous processors to improve the efficiency of model inference based on the Apache TVM compiler. However, TVM needs to specify the target hardware during the compilation process, which shifts the burden (decision problem) to the user (programmer). Wu et al. proposed a heterogeneous pipelined processing scheme on a multicore platform that divides the network into processing stages assigned to groups of cores [23]. In this work, most of the CNN computational operations are assigned to the GPU, and then the last layer of the network is offloaded to the CPU. The similarity between the above two approaches is that both of them assign a small portion of the model layer computations to the CPU, and offload other computing tasks to the GPU to improve the efficiency of model inference. However, this type of model offloading relies on the experience of experts and often requires analysis of the specific model structure, and has limited generalization capabilities for different models.

Some approaches propose a way using dynamic allocation. Jia et al. proposed CoDL, a concurrent deep learning inference framework for CPU and GPU on mobile devices [11], to guide the appropriate division of arithmetic computation tasks by building a latency predictor for different processors. Xu et al. proposed an online co-scheduling framework based on deep reinforcement learning, called COSREL, which utilizes deep reinforcement learning to balance inference latency, throughput, and energy efficiency. All computational resources on heterogeneous hardware are utilized to achieve significant speedups [24]. The above studies leverage heterogeneous processing capabilities among modules within integrated architectures to develop algorithms to efficiently distribute the computation tasks of model inference. Inspired by these ideas, this article divides the workload among different processors to accelerate model inference on a device with CPUs/GPUs.

Zeng et al. proposed a C^2RM framework to improve the parallelism between CPUs and GPUs by distributing workloads [26]. In their work, balancing the load between CPUs and GPUs improves the performance of model inference and saves energy. Saba et al. provided a systematic heterogeneous co-computing model, covering aspects such as CPU and GPU resource allocation and power budgeting, which utilizes machine learning to predict the performance of the collaborative solution and promotes collaborative optimization [18]. Aghapour et al. designed a CPU-GPU pipeline for CNN inference, which achieves network partitioning for CPU-GPU pipeline scheduling by running the **ARM Compute Library** (**ARM-CL**) on both ARM CPUs and Mali GPUs [1]. This approach improved the inference throughput by 75.88% on the Khadas Vim3 embedded platform equipped with Amlogic A311D HMPSoC. The above approaches explore CPU-GPU collaborative

computing to enhance inference performance and energy efficiency. These approaches demonstrate significant improvements—such as workload balancing, ML-driven resource allocation, and pipeline scheduling.

To sum up, existing methods accelerate model deployment tasks from the perspectives of deep learning model offloading, underlying hardware resource prediction, and CPU-GPU collaborative computing. Inspired by these works, this article analyzes the parallelism between CPU multithreading and CUDA streams that load GPU's computing tasks, and proposes a thread-level CUDA stream scheduling method that significantly improves the model inference speed.

3 Method

In this section, this article first introduces the Jetson Xavier NX embedded device with integrated CPU/GPU architectures, the model optimization for DNNs, and the process of model inference on the embedded device. Then, this article analyzes three common memory management strategies and the overhead analysis of managed memory strategy.

3.1 Notations and Preliminaries

3.1.1 Integrated Embedded Device. NVIDIA Jetson Xavier NX is an embedded device integrated with heterogeneous accelerators. NX's motherboard is equipped with a six-core ARMv8 CPU with a maximum frequency of 1.4GHz and a Volta GPU containing 384 CUDA cores. The GPU comes with 8GB of LPDDR4x memory, offering a bandwidth of 51.2GB/s. NVIDIA designs the NVDLA (NVIDIA Deep Learning Accelerator) as an open-source hardware architecture specifically for deep learning inference in edge devices. Relying on it, the theoretical peak performance of the GPU can reach 21 TOPS in the INT8 data format. Therefore, Jetson Xavier NX may be adopted as a compact platform for edge computing. The NX's NVDLA hardware is optional and needs to be set manually. However, NVDLA supports the vast majority of common **convolutional neural network** (**CNN**) architectures but has limited support for Transformer architectures, it cannot fully support the ViT model structure used in this article. Therefore, NVDLA module is not involved in this work.

The integrated architecture encapsulates the GPU within the CPU chip, known as an iGPU. The primary advantage is that the iGPU and CPU share a unified physical memory, allowing for in-place data processing without complex data transmission. This further reduces communication overhead. Therefore, iGPUs achieve higher energy efficiency compared with discrete GPU. Additionally, due to the absence of extra hardware resources (such as an I/O Hub), the power consumption of the iGPU is relatively lower, which enables integrated architectures to achieve a better balance between overall power consumption and performance. Although the integrated architecture meets the computing and storage resources required for deploying a PyTorch model, the inference speed is not fast enough for the integrated environment. TensorRT is used to compress PyTorch models to enhance inference performance, while the quantization process paradoxically reduces GPU utilization on integrated devices. Counterintuitively, this drop in resource utilization becomes a bottleneck, ultimately resulting in slower inference speeds. Therefore, this article aim at improving the GPU utilization of an integrated environment such as the NVIDIA Jetson Xavier NX, whose architecture is depicted in Figure 4. The proposed method and experiments are conducted on the NVIDIA Jetson Xavier NX.

3.1.2 Model Optimization. This article aims to improve the GPU utilization during deep learning model inference on a very specific platform, especially for a compressed model optimized by TensorRT. First, the input deep learning model is quantized by the post-training quantization method [17] to obtain the compressed model. Then, the compressed model is optimized by TensorRT to accelerate the model inference from the perspective of system. TensorRT is a C++ library for

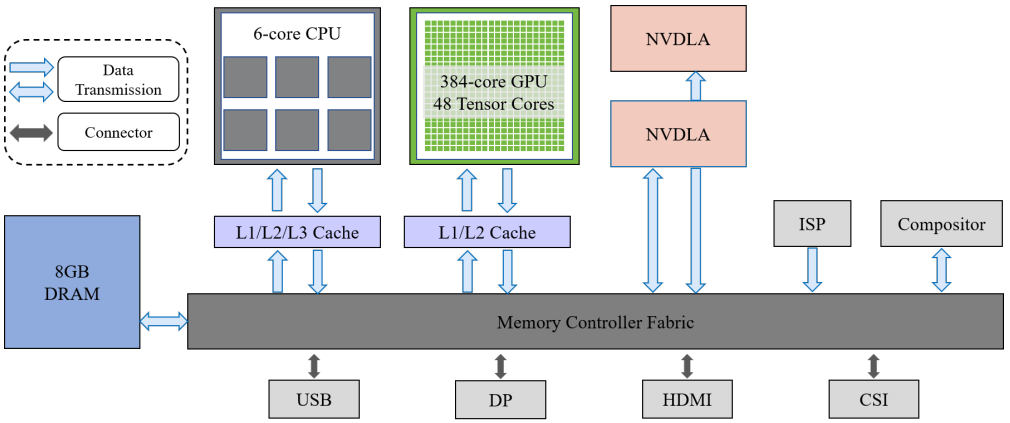


Fig. 4. The architecture diagram of the Jetson Xavier NX.

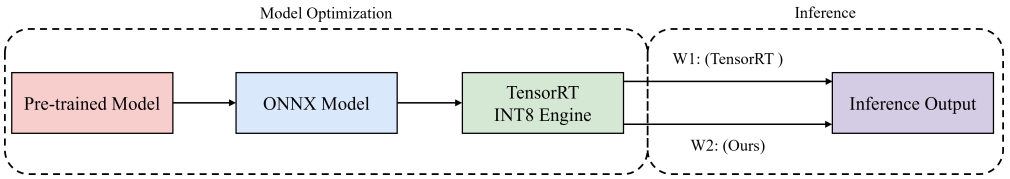


Fig. 5. A workflow for accelerating the inference process of a deep learning model.

accelerating the inference process on NVIDIA GPUs, which leverages the hardware characteristics of the **NVIDIA GPU and deep learning accelerator** (**NVDLA**). Given a pre-trained PyTorch model trained on a server, the optimization process needs the definition files of the model architecture and the model's weights files. The model architecture provides model parameters such as the number of layers, size, and various types of layer within the neural network.

As shown in Figure 5, in order to accelerate model inference, this article first optimizes the pre-trained model, which consists of two steps: transforming the pre-trained model into ONNX files and converting the model parameters' datatype from FP32 into FP16 or INT8 format by building TensorRT engine for the model. In detail, a PyTorch model can be exported to an ONNX file by using the torch.onnx.export() function in the PyTorch library. The torch.onnx.export() function accepts an input tensor to run a model that traces its execution, and then the traced model is exported to an ONNX file. Then the ONNX files is converted as a TensorRT engine by TensorRT API. This phase includes creating the network definition, importing the model through the ONNX parser, and building the TensorRT engine. The second step aims to further accelerate the inference process. The datatype of the DNN's parameter can be converted from FP32 to INT8 or FP16 to reduce model parameters. These operations need to be carried out manually.

3.1.3 The Process of Model Inference. After getting the TensorRT engine file, our proposed solution employs the TensorRT API to achieve model inference. As depicted in Figure 5, this article compares our proposed solution (W2 flow in Figure 5) with a state-of-the-art baseline (W1 flow). The W1 flow performs the inference process using TensorRT's default scheduling method, while the W2 flow performs the inference process using the method proposed in this article. There are six execution steps in the W1 flow: (1) creates an inference execution context; (2) allocates memory for inputs and outputs on the GPU; (3) the input data is transferred from the CPU to the allocated input

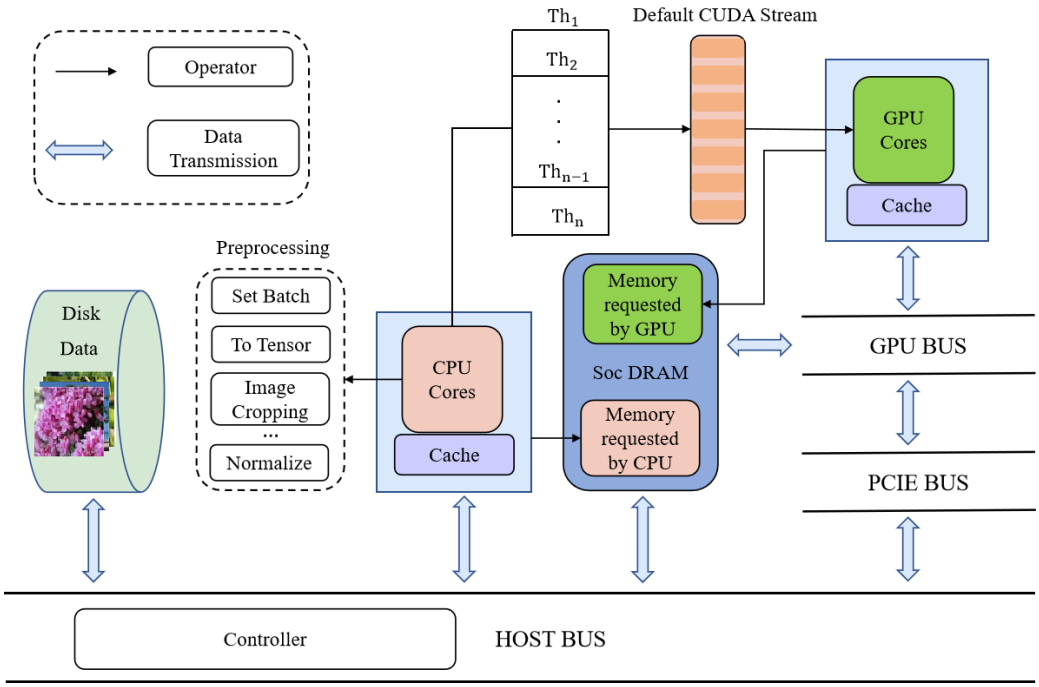


Fig. 6. The data flow of the inference process for a Pytorch model on Jetson Xaiver NX.

memory on the GPU; (4) executes the TensorRT engine inference using the asynchronous execution API; (5) the outputs are transferred back to the CPU memory, and (6) streams are synchronized for the data transmission and inference execution streams to ensure that all operations are completed. The executing steps of the W2 flow are similar to the W1 flow: (1) creates an inference execution context; (2) allocates unified input and output memory for the CPU and the GPU; (3) CPU removes the data from disk and preprocesses it; (4) uses asynchronous execution API to execute TensorRT engine inference; (5) synchronizes the streams used for the inference execution, and (6) CPU carries out the post-processing of the data, such as accuracy statistics.

The model inference process in the integrated architecture is depicted in Figure 6. Firstly, the image data stored on disk or other external storage devices will be migrated to DRAM for subsequent computational convenience. Then the input data of a PyTorch model will undergo a series of processing stages, such as transforming the data format into NumPy arrays or PyTorch tensors, a step usually performed in the on-chip CPU. Due to the inference computation of the model that usually runs on the GPU, data of a PyTorch model must be transferred from the CPU to the GPU, which can be accomplished by calling API methods such as `.to(device)` or `.cuda()`. The transmission process is implemented through a series of transfer buses (e.g., Host BUS, PCIE BUS) to a newly opened storage space. After the transmission process finished, TensorRT framework performs the computation tasks by utilizing internal API to call CUDA kernel using CUDA stream. The computation tasks will be added in a default CUDA stream and are performed by calling the on-chip GPU-cores. Once a computation task is completed, the data must be transferred from the GPU back to the CPU for further processing or saving the results. In the thread scheduling flow between CPU and GPU, the TensorRT framework minimizes the impact of inter-processor communication latency by utilizing the default CUDA stream to perform operations asynchronously

on the GPU. However, the cost of cross-thread communication and synchronization may offset these performance benefits of the optimization, leading to lower GPU utilization when inferring models on the integrated embedded devices. In order to reduce the inter-processor communication overhead, this article adopts the managed memory strategy as the memory management strategy to allow the CPU and GPU to share the same block of memory, which spares a fraction of data transfer and communication, thus improving the overall processing efficiency.

3.2 Memory Management Method

3.2.1 Memory Management Strategy. The memory management strategy aims to ensure that all the programs in the system can access the necessary memory without incurring memory conflicts or leaks. The management of GPU memory is used for memory allocation, data transfer, and memory release between the CPU and GPU during the inference process. The operations involved mainly include memory allocation, memory reclamation, memory protection, memory mapping, and which need to be managed by the operating system or application programs. Existing GPU's memory management strategies can be roughly divided into three categories: **device memory strategy** (**DMS**), **managed memory strategy** (**MMS**), and **pinned host memory strategy** (**PHMS**). These strategies differ internally in terms of memory consistency and data communication between the host and the device, determining the organization structure and data access methods of GPU memory. The pinned host memory strategy has the worst performance because the CPU cache is disabled during GPU operation, leading to further performance degradation for workloads executed on the CPU side. Therefore, this article will not mention the PHMS strategy any further.

The DMS is the default memory management strategy for the TensorRT framework during model inference. Figure 6 shows how the TensorRT framework uses DMS to allocate and manage memory on a unified memory architecture. For TensorRT, before data transfer, twice the amount of memory must be allocated for the same data, since it is necessary to create an equivalent memory space for the CPU or GPU. During data transfer, the on-chip GPU will access and process the data to perform computation tasks and return the computation results to the CPU for the next logical processing step. The explicit host-device synchronization (e.g., cudaMemcpy) required by the DMS introduces significant overhead, causing the GPU computational resources to remain under utilized due to frequent pipeline stalls. To alleviate this situation, our solution adopts the MMS.

In the MMS, memory allocation and deallocation are handled automatically by a runtime system or framework, rather than explicitly by the programmer, which is suitable for the device with a unified memory architecture, such as Jetson Xavier NX. The unified memory architecture is a storage architecture shared by both the CPU and the GPU. During the inference process of a PyTorch model using a CUDA program, the MMS defines a manageable memory space for specific use, in which all processors access a single coherent memory image with a common address space. Compared with DMS, MMS can simplify GPU programming by unifying the memory space of all GPUs and CPUs in the system, so that the CPU and the GPU do not need to request the same size of memory in their respective memories. And MMS eliminates the process of frequent back-and-forth data communication between processors. If MMS is used for model inference, the DRAM of the SoC will only allocate a piece of memory with contiguous address during the memory allocation process, which is jointly managed by the CPU and the GPU. Moreover, MMS eliminates the tedious step of explicitly copying data through APIs such as cudaMemcpy(), effectively reducing the load on the PCIe bus. The ease of use of globally shared data is improved by mapping data to the device memory address space, enabling the GPU to seamlessly access the region. Therefore, the work described in this article uses the managed memory strategy as the MMS used by the subsequent CPU and GPU scheduling algorithm.

3.2.2 Overhead Analysis of Managed Memory Strategy. This section analyzes the impact of MMS on the hardware performance during the process of model inference. The overhead is measured as the total GPU response time (including all memory management processes) minus the pure GPU execution time of the model inference. Using the MMS in iGPU systems requires ensuring data consistency between the host CPU and GPU [6]. Therefore, although it is possible to avoid frequently copying of memory, the overhead associated with address mapping and cache maintenance also needs to be considered, as these operations occur during the startup and completion phases of the kernel. The time overhead analysis of mapping and maintenance operations as follows.

Firstly, the overhead analysis is conducted for the kernel startup process. According to the MMS, whenever the host CPU modifies the allocated managed region, the corresponding data in the unified memory becomes outdated (without cache refresh). In such cases, if a kernel (denoted $K_i \in \mathcal{K}$) starts executing on the GPU and initiates memory access to the allocated memory region, a forced cache refresh between the host CPU and GPU is necessary to ensure data consistency (host cache refresh). This cache operation must be completed before the GPU can actually access the data. Therefore, the overhead can be inferred by analyzing the latency of page allocation and mapping on the GPU, as well as the frequency of cache refreshes on the CPU. Let T_i represent the overhead during kernel startup, which can be obtained through

$$T_i = \sum_{j=0}^{N_c-1} \left(S_j \times l_{mapping}(j) + C_{a_{\text{cpu}}} \right), \tag{1}$$

where S_i represents the size of the data accessed by K_i (which needs to be mapped from the CPU to the GPU memory). N_c denotes the size of the set $l_{mapping}$, which is the time cost per byte to create and map pages on the GPU. $C_{a_{\text{cpu}}}$ represents the cache refresh overhead from the CPU to the GPU, which is considered a constant, as it is calculated based on the time required to refresh all CPU caches to the main memory (to provide an upper limit).

Secondly, an analysis of the performance overhead is conducted upon kernel completion. Since managed data are considered local to the GPU during kernel execution (as CPU access to this memory area is generally disabled while the kernel is executing), managed area data accessed by kernels on the GPU will be considered outdated due to memory caching upon kernel completion. To ensure data consistency, data modified in the GPU cache must be written back to the main memory after kernel operations are completed. Let T_s denote the overhead caused by the required synchronization. To account for the worst-case scenario, assume that all cache lines used for data access will be invalidated upon kernel completion, and the overhead of writing all data back to memory can be represented as a constant $C_{a_{gpu}}$. Therefore, T_s can be derived as

$$T_s = Min\Big(\big(\sum_{j=0}^{|\mathcal{K}|-1} B^j_{\text{write}} \times l_{mem}(j) \big), N_k \times C_{a_{\text{gpu}}} \Big), \tag{2}$$

where B^j_{write} represents the size of the data modified by the kernel K_i, and N_k denotes the number of kernels in the application. In summary, the per-occurrence overhead C_M of the application can be expressed as the sum of T_i and T_s as

$$C_M = T_i + T_s. \tag{3}$$

The data flow in real applications often consists of multiple batches. Therefore, the total overhead O_M during a complete inference process of a deep learning model can be calculated by

$$O_M = \sum_{j=1}^{E_p} C^j_M. \tag{4}$$

It can be observed from Equation (4) that the overhead of the MMS policy depends on the data size (i.e., B^{j}_{write} item), and the overhead stems mainly from the overhead due to memory migration or page errors as described in Equation (2). Moreover, the overhead of MMS also depends on N_k (the number of kernel starts), which suggests that applications that require multiple kernel starts (e.g., Gaussian applications) are negatively affected. Considering that the MMS policy is theoretically effective in reducing inter-processor communication, this article adopts it as the memory policy employed during CPU and GPU collaboration. And by modeling the overhead, it also guides the subsequent method.

3.3 Pipeline-Style Heterogeneous Stream Scheduling Strategy

In order to fully leverage the NX's hardware resources, combining the advantages of heterogeneous processors, this article designs the CPU and GPU collaborative scheduling algorithm based on unified memory. Firstly, the tasks of model inference are divided into the CPU and GPU: let the CPU handle data-related preprocessing and post-processing, and take charge of coordinating the scheduling tasks; the GPU is responsible for performing the actual inference computations. Secondly, a pipeline-style scheduling strategy is presented to effectively overlap the data processing using CPU and the computing time through parallel execution of tasks using GPU. The unified memory is utilized to reduce communication overhead between CPU and GPU.

To implement the above ideas, there are two challenges must be addressed: 1, how to prevent race conditions when multiple threads compete for shared GPU resources. 2, how to resolve incorrect GPU computation results caused by concurrent unified memory access across multiple kernels. The former challenge arises as a consequence of threads' preemption in CPU when the GPU executes computational tasks, terminates a current task, and switches to tasks from other batches. The latter refers to a computational inconsistency where the GPU correctly executes a task, but the results are incorrectly mapped to unrelated memory locations due to improper synchronization, leading to erroneous aggregation during host-side data collection. Thus, our solution binds the multiple threads of CPU with multiple CUDA streams of GPU to address these issues as shown in Figure 7. For the first issue, race conditions when multiple threads access shared GPU resources can be prevented by placing GPU kernel functions within critical sections and explicit synchronization enforced through coordinated multi-threading and CUDA stream management. The execution process completes by going through the following five stages: (1) The CPU core is used to complete data preprocessing and trigger the scheduling of CUDA stream; (2) Upon encountering a critical section that one thread accesses the critical resource (i.e., the GPU), blocking the remaining threads and adding the execution task to the waiting set; (3) Within the critical section, the GPU acquires data and performs specific computations; (4) Exiting the critical section, adding the computed result into the data post-processing set, and waiting for the corresponding thread to retrieve the data and perform post-processing tasks; (5) Repeating the above steps until all data has been processed. More details are shown in Algorithm 1 and Figure 7.

Figure 7 depicts the overall workflow for model inference. The parallel pipeline scheduling design includes data preparation and inter-stream scheduling. Data preparation involves how multiple threads concurrently capture and partition data. Firstly, the test dataset can be divided into multiple subsets. Let D_i represent the part ith of the test dataset, then the dataset will be divided as: D_1, D_2, and D_3. A subdataset will be divided into j batches during the data preparation process, where $D_{i,j}$ represents the jth data slice of the ith subdataset. Then, the data will be preprocessed and assigned address information. Assume that three threads are set for parallel scheduling in the execution process, Th_i represents the ith thread, each dataset subset is captured by a corresponding thread. According to the address information, the CUDA stream scheduling information for the corresponding thread is mapped for the following scheduling execution.

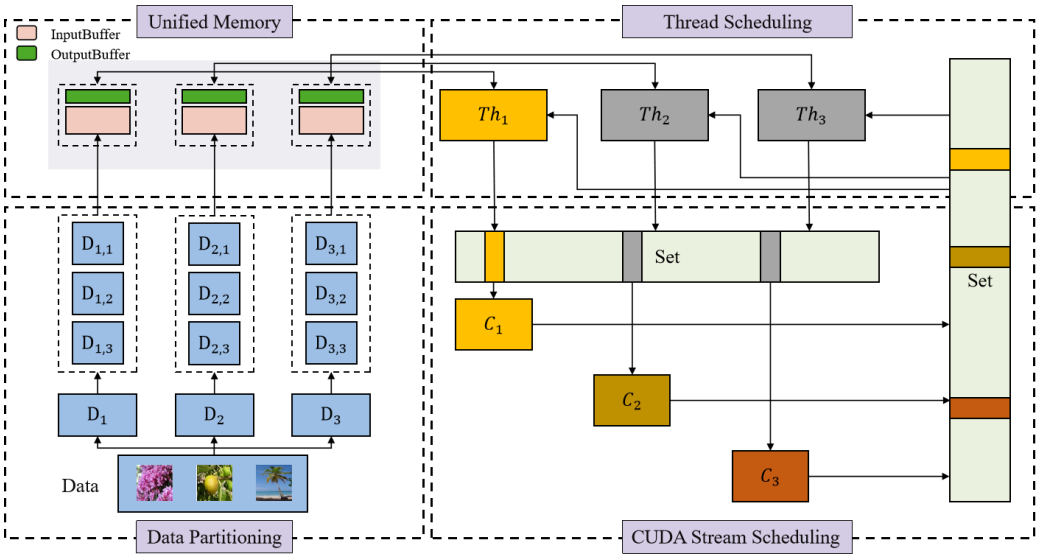


Fig. 7. Parallel pipeline scheduling design drawing.

ALGORITHM 1: A Multi-Thread Stream Scheduling Algorithm Based on Unified Memory

INPUT: TensorRT serialized engine file: M', Thread of execution: th, Executing CUDA stream: *Stream*, Thread Pool: *Thread*, Unified Memory input buffer: $inBuffer$, Unified Memory output buffer: $outBuffer$

OUTPUT: Result

Load and deserialize the TensorRT engine M'.
Create CUDA execution contexts based on the number of threads (Num(*Thread*)) available in thread pool.
Prepare the input image data and labels.
for i=0;i<Num(*Thread*);i++ **do**
    Allocate input buffers $inBuffer$ and output buffers $outBuffer$ for images with fixed batch size based on CPUGPU unified memory.
    Create multiple CUDA streams, bind(th_i -> $Stream_i$).
end for
for Parallel do
    Associate(*th* -> $inBuffer$), accessed by the CPU.
    CPU loads image data and labels.
    Synchronize *Stream*.
    if critical then
        Associate(*th* -> $inBuffer$ and $outBuffer$), accessed by *th*.
        GPU executes inference tasks on *th*.
        Synchronize *stream*.
        Associate(*th* -> $outBuffer$), accessed by the CPU.
        Synchronize *stream*.
    end if
    CPU statistics precision data.
end for
Free $inBuffer$, $outBuffer$, and *stream*.

The process of inter-stream scheduling aims to control the concurrent execution of CUDA streams, which depends on the task set for CUDA streams and critical sections for avoiding uncontrolled preemptions in the CPU. Let C_i represent the ith CUDA stream, Q represents a set for these computing tasks (Set in Figure 7) which are waiting to be executed, and L represents a disordered set for completed inference tasks, and it is waiting for returning data. According to the number of threads, the scheduling of inter-stream can be divided into multiple subtasks with their corresponding information for a CUDA stream, respectively. Every subtask with the corresponding information for a CUDA stream is encapsulated with the address information of a slice of data (such as $D_{1,1}$). When any idle threads, a subtask will be added to the set Q and wait to execute. Then the task is executed, a corresponding CUDA stream C_i will be triggered and used to complete the computing task according to the CUDA stream scheduling information. If the computing task is completed, the computed results will wait for returning to the thread to perform data post-processing. After the thread completes the data post-processing task, it will immediately take the next round of slice data and repeat the above operation until all slice data are executed. In the above process, the critical section ensures that the thread executing each subtask is not preempted, thus allowing the subtask to run to completion without interruption. Hence, the statistical calculation result for data post-processing is correct.

For the second problem, to resolve incorrect GPU computation results caused by concurrent unified memory access across multiple kernels, the work described in this article maps each thread to a corresponding CUDA stream and ensures that the association is not changed during the thread life cycle. As shown in Figure 7, thread Th_1 is associated with the CUDA stream C_1. In addition, the memory bytes are associated with the specified stream through APIs such as cuda_Stream_Attach_MemAsync(), and then the CUDA stream is controlled by the thread. The other streams are not allowed to access the memory data. By associating the allocation thread with a specific stream, the CUDA program can ensure that only the kernel launched into the stream will access the intended data. And the CUDA program allows managed allocations to be explicitly associated with the CUDA streams. In this way, the use of the kernel for the data can be indicated based on whether the kernel is launched into a specified stream. Moreover, the unified memory system allows other threads to access this memory area as long as all tasks in the corresponding CUDA stream are completed, regardless of whether other streams are active. In fact, this limits the exclusive ownership of the managed memory area by the active GPU to each stream activity, rather than the entire GPU activity. In summary, by binding thread with CUDA stream and memory bytes with CUDA stream (as mentioned in Section 3.2.1), the CPU core can only interact with the corresponding CUDA stream, thereby the processing tasks are aligned between the CPU and GPU. This solution can be summarized as follows: (1) establish a mapping relationship between threads and CUDA streams; (2) associate the corresponding memory region with the specified stream; (3) schedule the threads automatically (when there are idle threads) to promote parallelism across CUDA streams.

In conclusion, the process of multi-thread stream scheduling Algorithm based on unified memory is as follows. *Algorithm 1* includes five stages.

(1) **Initialization Stage (Lines 1–7):** The computing environment is prepared based on the NVIDIA unified memory. Within the OpenMP environment, the deserialized TensorRT engine creates the corresponding number of CUDA execution contexts, and then the corresponding CUDA stream is created and bound for each thread. At the same time, input and output buffers are allocated for each batch of images to be processed on unified memory in preparation for data storage and inference calculations.

(2) **PreProcess Stage (Lines 9–11):** In this parallel section, each thread loads image data from the CPU into unified memory. Synchronization ensures all threads complete data loading before proceeding.

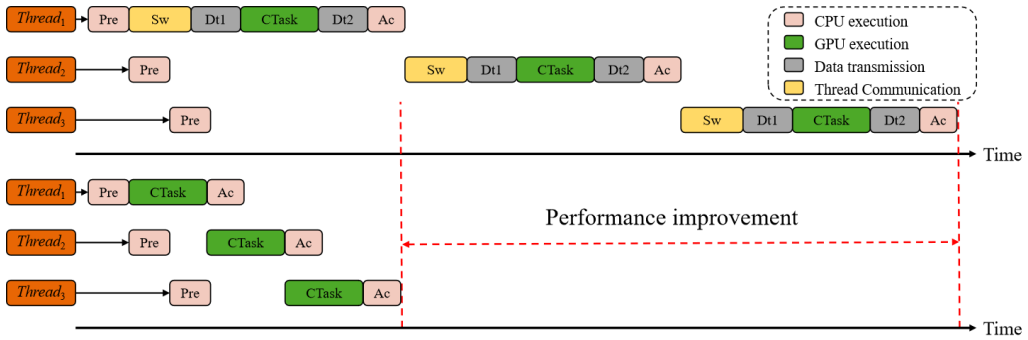


Fig. 8. The operation flow of the inference process for the TensorRT's scheduling method and the proposed parallel pipeline scheduling method.

(3) **Inference Stage (Lines 13–17):** This is a critical section where only one thread executed at a time. Each thread attaches its stream, launches GPU inference, and synchronizes to ensure correctness and avoid data races.

(4) **Postprocess Stage (Line 19):** The CPU performs multi-thread aggregation of results.

(5) **Release Stage (Line 21):** Finally, all CUDA resources (streams and buffers) are deallocated to release memory.

There are additional notes for Algorithm 1: (1) The DNN model M' is the serialized engine file, which was obtained by adopting the TensorRT to optimize the PyTorch pre-trained model. The details of optimization process are mentioned in Section 3.1.2. The engine file is deserialized to obtain the CUDA execution context; (2) The variable *th* represents the thread executed, whose number is specified before the program is executed. The number of threads is consistent with the number of CUDA streams, since the CUDA streams need to be mapped one-to-one with the threads; (3) The sizes of thread pool *Thread* is constrained to a maximum of six threads to align with the hexa-core architecture of the NX device's CPU.

From a theoretical perspective, for the DNN model M, the performance of the inference process in NX can be analyzed in a way of operation flow, and the process is depicted in Figure 8. As shown in Figure 8, the upper part shows the scenario of using the default scheduling method of TensorRT framework with DMS strategy to execute tasks. As mentioned in Section 3.2.2, during the inference process, a modest kernel overhead is incurred regardless of any memory management strategies. But the overhead is so small that it can be ignored [7], and it is not enough to affect the overall performance of the inference process. Therefore, for the default inference process in TensorRT, data execution operation mainly includes six parts: data preparation (Pre), thread switching and the binding of the CUDA stream (Sw), data transfer from CPU to GPU (Dt1), GPU inference calculation (CTask), statistics of the results (Ac), and data transfer from GPU to CPU (Dt2). The lower part of Figure 8 shows the scenario of using the proposed parallel pipeline scheduling method with MMS strategy to execute tasks. Different from the TensorRT's default method, the proposed method establishes a mapping between threads and CUDA streams before executing specific inference tasks, and no additional overhead will be generated during the execution of CUDA streams due to communication switching between multiple threads. It is obvious that the lower half of Figure 8 has no performance overhead for thread switching and the binding of the CUDA stream. Moreover, the MMS strategy reduces the data transmission operations between CPU and GPU (Dt1 and Dt2). In summary, the proposed method only retains two operations of Pre and CTask in practice. Assume that the number of threads is 3 for the TensorRT's default method and the proposed method during the inference process. The number of data batches is m. The execution time of Pre is Δt_p. The

overhead time of Sw is Δt_s. The overhead time of Dt1 and Dt2 is Δt_{d1} and Δt_{d2}, respectively. And the execution time of inference task (CTask) is Δt_c. Moreover, the precision statistics time by using the CPU is Δt_a. Then the time analysis model for the TensorRT's default method and the proposed method on NX can be respectively established as

$$T_1 = \Delta t_p + \sum_{i=1}^{m} (\Delta t_s + \Delta t_{d1} + \Delta t_c + \Delta t_{d2} + \Delta t_a), \tag{5}$$

and

$$T_2 = \Delta t_p + \sum_{i=1}^{m} (\Delta t_c) + \Delta t_a, \tag{6}$$

respectively.

In Equation (5), the cross-thread communication and synchronization cannot run in parallel with precision statistics since all computing tasks in a single CUDA stream must be addressed strictly sequentially. And the overhead of Sw is greater than the overhead time of Dt2 or the overhead of Sw is equal to the overhead time of Dt2 (i.e., $\Delta t_s \geq \Delta t_{d2}$). In Equation (6), since the computational complexity of model inference (CTask) is much greater than the computational complexity of CPU post-processing (Ac), Δt_c is much greater than Δt_a, while Pre and Ac can run in parallel with CTask. In summary, the final results presented by TensorRT and the final results presented by the proposed method should be as shown in Figure 8 under ideal inference conditions. It can be seen that after pipeline processing, the model inference performance is significantly improved.

4 Experiments

In this section, we report extensive experiments to discuss the efficiency and scalability of the proposed approach. Firstly, we describe the setup of the experimental environment, the details of publicly available models, and the datasets. Then we test several metrics to present the inference performance on the Jetson Xavier NX device. Finally, we compare the inference performance between the proposed method with the default method in TensorRT and present the corresponding analysis to demonstrate the superior performance of the proposed method.

4.1 Experimental Details

Configuration: the hardware and software configurations are all listed in Table 1. We use the Nvidia Jetson Xavier NX with Volta GV10B GPU and Carmel ARM®v8.2 CPU as the basic experimental platform. The size of memory RAM is 8GB. For deploying deep learning models, we install Ubuntu 18.04.2 LTS as the operating system and adopt PyTorch deep learning framework with version of 1.12.0. We adopt TensorRT as optimization framework with version of 8.5.1. We install CUDA as the development environment for GPU with version of 11.2.

Models and Datasets: we conduct extensive experiments on two classification tasks. Seven networks as VGG19, Xception, GoogLeNet, ResNet34/101, WideResNet, and ViT-Tiny are tested on two publicly available datasets CIFAR-100 and Mini-ImageNet, these details are shown in Table 2, respectively. CIFAR-100 is a classic image classification dataset commonly used in computer vision research and algorithm evaluation. This dataset contains 60,000 images, each with a resolution of 32x32 pixels and RGB channels. The CIFAR-100 dataset finely includes images of 100 different categories, with each category comprising 600 images. In addition, Mini-ImageNet is a commonly used small-scale image classification dataset. It is a subset selected from the ImageNet dataset, containing images from 100 categories, with 600 images per category, totaling 60,000 images. Each image has a resolution of 224x224 pixels and RGB channels. Although Mini-ImageNet is the smaller version of the ImageNet dataset, which contains a certain level of complexity and diversity.

Table 1. The Experimental Environment on Jetson Xavier NX

System	Modules	Details
Hardware	GPU	NVIDIA Volta GV10B
	CPU	NVIDIA Carmel ARM®v8.2
	Memory RAM	8GB
Software	Deep learning framework	PyTorch 1.12.0
	Model optimization framework	TensorRT 8.5.1
	CUDA version	CUDA 11.2
	Operating system	Ubuntu 18.04.2 LTS (64-bit)
VM	Memory RAM	12GB
	Kernel	Linux 4.3.0
	Operating system	Ubuntu 18.04.2 LTS (64-bit)

Table 2. Test Model Information

Models	Top-1(%)	Storage(MB)	Parameters(M)	MACs(G)	Datasets
ViT-Tiny	58.25%	61.69	20.62	0.35	CIFAR-100
ViT-Tiny*	56.53%	**6.88**			
GoogLeNet	75.2%	24.64	6.40	0.54	CIFAR-100
GoogLeNet*	75.3%	**7.27**			
VGG19	72.2%	76.50	20.08	0.40	CIFAR-100
VGG19*	72.0%	**19.70**			
ResNet34	75.8%	81.50	21.33	1.16	CIFAR-100
ResNet34*	75.8%	**21.10**			
Xception	76.3%	80.47	21.01	1.14	CIFAR-100
Xception*	73.9%	**21.80**			
ResNet101	77.0%	163.49	42.70	2.52	CIFAR-100
ResNet101*	76.0%	**30.30**			
WideResNet	77.9%	213.48	55.91	8.09	CIFAR-100
WideResNet*	77.7%	**54.20**			
ViT-Tiny	53.26%	65.72	21.89	1.10	Mini-ImageNet
ViT-Tiny*	51.11%	**7.13**			
GoogLeNet	75.2%	24.64	6.40	0.54	Mini-ImageNet
GoogLeNet*	77.12%	**7.27**			
VGG19	74.9%	76.50	20.08	0.40	Mini-ImageNet
VGG19*	80.12%	**19.70**			
ResNet34	75.1%	81.50	21.33	1.16	Mini-ImageNet
ResNet34*	80.64%	**21.10**			

Baseline: we optimize seven networks with model quantization [17] and operator fusion by TensorRT, the details of models are shown in Table 1. The data bit of quantization is INT8. Then, these optimized models are as baselines for comparing the TensorRT's default method and the proposed method in this article.

Table 3. Latency Tests are Performed with Different Models and Batch Size, and the Test Dataset is CIFAR-100

Batch Size	VGG19*		GoogLeNet*		Xception*		ViT-Tiny*		ResNet101*		WideResNet*	
	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT
1	11.9	95.9	14.9	111.0	15.8	100.7	26.98	198.01	64.2	190.6	32.6	117.2
16	1.4	9.1	2.8	11.9	4.3	12.3	10.59	24.23	32.5	70.1	16.7	22.2
32	1.1	6.8	2.3	6.9	3.9	7.8	10.14	21.20	31.4	65.9	16.0	19.3
64	0.9	5.0	2.1	5.6	3.8	6.4	8.85	20.32	30.9	64.4	15.8	18.1
128	0.8	4.6	2.0	4.8	3.7	6.2	–	–	–	–	–	–

TRT refers to the default scheduling method in TensorRT.

4.2 Evaluation Metrics

In this article, we test six models on the CIFAR-100 dataset and select VGG19*, GoogLeNet*, ViT-Tiny*, and ResNet34* for verification on the Mini-ImageNet dataset to illustrate the scalability of the proposed method. The experimental metrics include inference latency (seconds, s), GPU utilization (%) (i.e., the proportion of time that the **stream processor** (**SM**) is active during the sampling period), GPU throughput (frames per second, FPS), memory usage (megabytes, MB) and energy consumption (joules, J). The throughput is obtained based on the elapsed time of the entire inference phase of the model, which is given by

$$\text{Throughput} = \frac{N}{Time}, \tag{7}$$

where N represents the number of input images during the inference process. The memory usage is the inference memory usage of the model during its inference task, which is obtained based on the memory change after inference stabilization and before model inference as follows:

$$\Delta M = M_{\text{a}} - M_{\text{b}}, \tag{8}$$

where ΔM represents the model inference memory footprint, M_a represents the device memory usage, which is recorded after the model inference phase stabilized, and M_{b} represents the device memory usage before the model inference. The energy consumption is the average energy consumption in total inference stage, which is calculated by

$$\text{Energy} = \left(\frac{P}{1000}\right) \times \text{Time}, \tag{9}$$

where P is the average power of the NX when the system achieves stable state.

4.3 Time Analysis During Model Inference

In order to illustrate the effectiveness of the proposed approach, we conduct experiments to test inference latency during model inference. In this section, we perform the inference process of six models on the CIFAR-100 dataset and four models on the Mini-ImageNet dataset under the proposed optimization method and the default method in TensorRT respectively. We test each model with multiple batch sizes within the affordable range of on-chip resources. The batch sizes are 1, 2, 4, 8, 16, 32, 64, and 128, respectively. The experimental results are displayed in Table 3 and Table 4, respectively.

Table 3 illustrates the total inference time of six models on the testset of the CIFAR-100 dataset, which shows that when using our inference acceleration method, the inference latency of the six models is significantly lower than the inference latency obtained using the default inference method of the TensorRT framework. Moreover, as the batch size increases, the inference time of the

Table 4. Latency Tests are Performed with Different Models and Batch Size, and the Test Dataset is Mini-ImageNet

VGG19*			GoogLeNet*			ResNet34*			ViT-Tiny*		
Batch Size	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT
1	47.8	255.9	1	104.6	313.1	1	122.2	297.9	1	53.70	214.17
8	38.0	176.3	2	96.3	248.6	4	110.5	251.4	8	38.58	72.88
16	37.1	170.7	4	91.8	223.4	8	107.2	237.9	16	37.88	76.00

TRT refers to the default scheduling method in TensorRT.

six models decreases significantly. However, the inference latency of these models by our inference acceleration method is still significantly lower than the inference latency obtained by the PyTorch framework. It should be noted that, for ResNet101*, ViT-Tiny*, and WideResNet*, when the batch size reaches 128, due to the memory limitation of the device, it is not possible to complete the conversion of the TensorRT model, and therefore these data are not listed.

Table 4 illustrates the total inference time of four models on the testset of the Mini-ImageNet dataset. Because of the image size increases (32×32 to 224×224), the on-chip computation resources are unable to support a larger batch size. Thus, VGG19* and ViT-Tiny* were tested with batch sizes of 1, 8, and 16; GoogLeNet* was tested at batch sizes of 1, 2, and 4; and ResNet34* was tested at batch sizes of 1, 4, and 8. As shown in Table 4, the inference latency of the three models obtained by our method is also significantly lower than the inference latency obtained using the default inference method of the TensorRT framework.

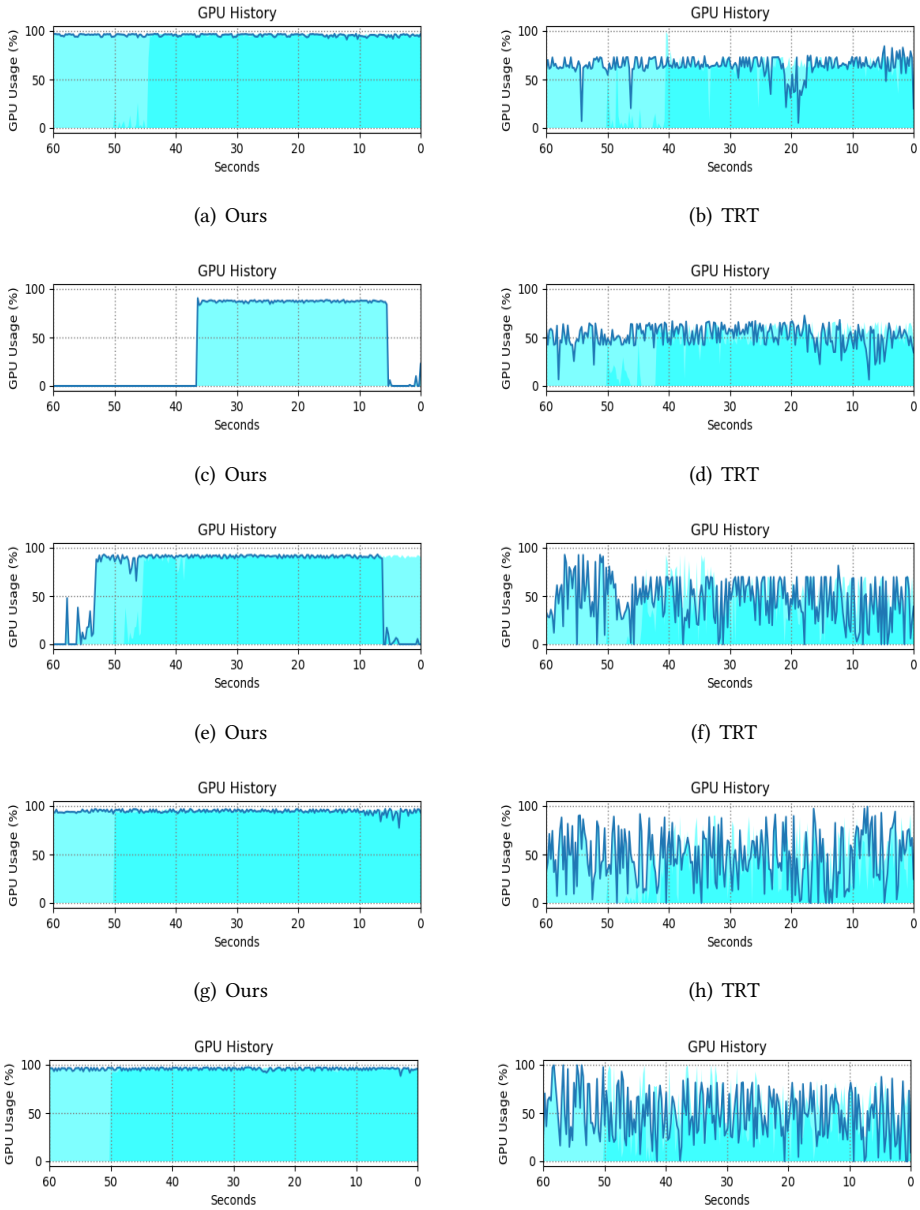
In conclusion, it is easy to see that the proposed method has a huge performance improvement over the TensorRT inference algorithm, especially with the batch size of 1, which meets the task requirements of real-time classification. Given a model, compared with the default method in TensorRT, the inference speedup ratio of our proposed method across different batch sizes can achieve around 2x-10x.

4.4 GPU Utilization Test During Model Inference

In this section, we visualize the GPU utilization during model inference to illustrate the effective utilization of on-chip resources by the proposed method. The inference process is performed under the proposed method and TensorRT's default method, respectively. ResNet101* and WideResNet* are performed on the CIFAR-100 testset. The VGG19*, GoogLeNet*, and ResNet34* are conducted on the Mini-ImageNet testset. Figure 9 shows that, for any model, it is obvious that the GPU utilization of our proposed method (a, c, e, g, i) is higher than using the default method in TensorRT (b, d, f, h, j) during the inference process with a batch size of 1. The proposed method significantly improves the GPU utilization in the model inference process, with the improvement ranging from 30% to 50%. And it can be seen from Figure 9 (c, e) that higher GPU utilization leads to a short inference time. Therefore, the proposed method in this article can effectively improve the GPU utilization and decrease the inference time. In addition, observing the fluctuation of GPU utilization shown in the figure, it is not difficult to see that the algorithm proposed in this article has better stability, which also shows that the algorithm in this article can realize the switching and scheduling between different threads more smoothly in the process of multi-batch task execution. In summary, the reasoning algorithm proposed in this article can more fully utilize the device resources.

4.5 Model Throughput During Model Inference

In this section, we discuss experiments conducted to demonstrate the impact of the proposed method on the model throughput. For that, we perform the inference process of six models on



(a) Ours (b) TRT

(c) Ours (d) TRT

(e) Ours (f) TRT

(g) Ours (h) TRT

(i) Ours (j) TRT

Fig. 9. The GPU utilization of the deep learning models when perform model inference using the proposed method and the TensorRT's default method, respectively. (a) and (b) perform ResNet101* on the CIFAR-100. (c) and (d) perform WideResNet* on the CIFAR-100 testset. (e) and (f) perform VGG19* on the Mini-ImageNet testset. (g) and (h) perform GoogLeNet* on the Mini-ImageNet testset. (i) and (j) perform ResNet34* on the Mini-ImageNet testset. The batch size of experiments is set to 1 during the inference process. TRT refers to the default scheduling method in TensorRT.

Table 5. GPU Throughput Tests are Performed with Different Models and Batch Size, and the Test Dataset is CIFAR-100

Batch Size	VGG19*		GoogLeNet*		Xception*		ViT-Tiny*		ResNet101*		WideResNet*	
	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT
1	2840.3	104.3	671.1	90.1	632.9	99.3	370.66	50.51	155.8	52.5	306.7	85.3
16	7142.9	1098.9	3571.4	840.3	2325.6	813.0	944.02	412.71	307.7	142.7	598.8	450.5
32	9090.9	1470.6	4347.8	1449.3	2564.1	1282.1	986.00	471.70	319.0	151.7	625.0	518.1
64	11111.1	2000.0	4761.9	1785.7	2631.6	1562.5	1130.45	492.13	324.1	155.3	632.9	552.5
128	12500.0	2173.9	5000.0	2083.3	2702.7	1612.9	–	–	–	–	–	–

TRT refers to the default scheduling method in TensorRT.

Table 6. GPU Throughput Tests are Performed with Different Models and Batch Size, and the Test Dataset is Mini-ImageNet

VGG19*			GoogLeNet*			ResNet34*			ViT-Tiny*		
Batch Size	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT
1	209.2	39.1	1	95.6	31.9	1	81.8	33.6	1	186.20	46.69
8	263.2	56.7	2	103.8	40.2	4	90.5	39.8	8	259.22	137.21
16	269.5	58.6	4	108.9	44.8	8	93.3	42.0	16	264.01	131.58

TRT refers to the default scheduling method in TensorRT.

the CIFAR-100 dataset under the proposed optimization method and the default configuration in TensorRT, respectively. The batch sizes are 1, 16, 32, 64, and 128, respectively. The experimental results are shown in Table 5. Moreover, we perform the inference process of VGG19*, GoogLeNet*, ResNet34*, and ViT-Tiny* on the Mini-ImageNet dataset under the proposed optimization method and the default configuration in TensorRT, respectively. The batch sizes for VGG19* are 1, 8, and 16. The batch sizes for GoogLeNet* are 1, 2, and 4. The batch sizes for ResNet34* are 1, 4, and 8. The batch sizes for ViT-Tiny* are 1, 8, and 16. The experimental results are shown in Table 6.

Tables 5 and 6 show that when using our inference acceleration method, the model throughput of all models is significantly higher than the model throughput obtained using the default inference method of the TensorRT framework. Due to the limitation of device resources, with the influence of the complexity of the model structure, there is a huge difference in model throughput. Moreover, as the batch size increases, the model throughput of all of these models increases. However, the model throughput of all these models by our inference acceleration method is still significantly higher than the model throughput obtained by the TensorRT's default method.

In conclusion, the inference algorithm proposed in this article has a huge performance improvement over TensorRT's default inference method, especially with the batch size of 1, which meets the task requirements of real-time classification. In addition, even on a slightly larger dataset such as Mini-ImageNet, after the scheduling and processing of this article's algorithm, the processing frame rate of the image can reach more than 100 frames. This fully demonstrates that the algorithm proposed in this article can better meet the needs of real life and industrial production.

4.6 Analysis of Memory Usage During Model Inference

In this section, to compare the impact of the proposed algorithm and the TensorRT's default method on on-chip memory, this article analyzes the memory usage of NX during the model inference process. For that, we perform inference tasks of six models on the CIFAR-100 dataset with batch sizes of 1, 16, 32, 64, and 128, respectively. The experimental results are shown in Table 7. Moreover, we

Table 7. Memory Footprint During Inference Test with Different Models and Various Batch Sizes, the Test Dataset is CIFAR-100

Batch Size	VGG19*		GoogLeNet*		Xception*		ViT-Tiny*		ResNet101*		WideResNet*	
	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT
1	252	443	246	465	250	467	439	973	508	705	249	439
16	246	450	246	462	252	461	442	974	254	451	243	446
32	265	461	246	460	249	471	443	978	270	347	251	450
64	250	455	246	464	234	478	446	969	365	719	234	449
128	251	460	268	435	248	485	–	–	–	–	–	–

TRT refers to the default scheduling method in TensorRT.

Table 8. Memory Footprint During Inference Test with Different Models and Batch Size, the Test Dataset is Mini-ImageNet

VGG19*			GoogLeNet*			ResNet34*			ViT-Tiny*		
Batch Size	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT
1	349	379	1	244	354	1	248	385	1	480	1452
8	263	393	2	253	373	4	260	387	8	486	1789
16	252	427	4	277	407	8	273	394	16	532	2143

TRT refers to the default scheduling method in TensorRT.

perform inference tasks of VGG19*, ViT-Tiny*, GoogLeNet*, and ResNet34* on the Mini-ImageNet dataset. Among them, VGG19* and ViT-Tiny* were tested at batch sizes of 1, 8, and 16, respectively; GoogLeNet* was tested at batch sizes of 1, 2, and 4, respectively; and ResNet34* was tested at batch sizes of 1, 4, and 8, respectively. These experimental results are shown in Table 8.

Table 7 reports the memory footprint of six models when performing the inference tasks on the testset of the CIFAR-100 dataset. Table 8 illustrates the model footprint of the VGG19*, ViT-Tiny*, GoogLeNet*, and ResNet34* when performing the corresponding inference tasks on the testset of the Mini-ImageNet dataset. From the Table 7, it is obvious that the setting of batch size has little influence on the memory footprint for different models during the inference process, the memory footprint is mainly related to the complexity of the model. Combining with the data in Table 8, it can be observed that the memory footprint is also related to the size of the dataset. It is obvious that the memory footprint of all these models by our inference acceleration method is still significantly lower than the memory footprint utilized by the TensorRT framework, the inference algorithm proposed in this article has a partial memory saving compared with TensorRT's default inference method, with the saving percentage ranging from 30% to 40%. This is due to the unified memory management strategy in our proposed approach. This also indicates that the proposed algorithm is more suitable for embedded devices, which have limited memory capacity.

4.7 Energy Consumption Statistics During Model Inference

Although the algorithm proposed in this article does not make any changes to the model, by speeding up the inference process, it can theoretically reduce the energy cost of some systems. Therefore, we analyze the energy cost of the model inference process in NX device, which is obtained by measuring the average power of the NX throughout the inference phase and combining it with the overall inference time in Section 4.3. In this section, we perform inference tasks of six models on the CIFAR-100 dataset with batch sizes of 1, 16, 32, 64, and 128, respectively. The

Table 9. The Inference Energy Consumption Test is Performed with Different Models and Batch Size, and the Test Dataset is CIFAR-100

Batch Size	VGG19*		GoogLeNet*		Xception*		ViT-Tiny*		ResNet101*		WideResNet*	
	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT	Ours	TRT
1	179.7	866.2	203.4	982.6	257.5	934.2	449.98	1922.48	1588.1	3785.5	680.4	1286.0
16	21.2	92.5	39.8	126.2	73.4	153.5	299.81	700.71	828.5	2011.3	455.8	475.6
32	13.5	71.6	33.8	78.7	64.4	110.4	303.70	671.85	809.6	1914.3	441.3	462.3
64	10.7	59.4	30.4	66.8	62.6	94.1	283.16	688.52	805.0	1882.4	437.6	457.0
128	9.2	50.8	27.9	62.1	62.9	88.9	–	–	–	–	–	–

TRT refers to the default scheduling method in TensorRT.

Table 10. The Inference Energy Consumption Test is Performed with Different Models and Batch Size, and the Test Dataset is Mini-ImageNet

VGG19*			ViT-Tiny*		GoogLeNet*			ResNet34*		
Batch Size	Ours	TRT	Ours	TRT	Batch Size	Ours	TRT	Batch Size	Ours	TRT
1	1468.4	2838.7	1364.29	3203.06	1	2957.6	4554.9	1	3983.7	5837.9
8	1151.2	2532.2	1206.57	2580.10	2	2759.8	4341.8	4	3424.8	4934.2
16	1061.5	2477.4	1229.03	2697.85	4	2720.1	4037.7	8	3232.8	4576.7

TRT refers to the default scheduling method in TensorRT.

experimental results are shown in Table 9. Moreover, we perform inference tasks of VGG19*, ViT-Tiny*, GoogLeNet*, and ResNet34* on the Mini-ImageNet dataset. Among them, VGG19* and ViT-Tiny* were tested at batch sizes of 1, 8, and 16, respectively; GoogLeNet* was tested at batch sizes of 1, 2, and 4, respectively; and ResNet34* was tested at batch sizes of 1, 4, and 8, respectively. These experimental results are shown in Table 10.

Table 9 reports the energy cost of six models when performing the inference tasks on the testset of the CIFAR-100 dataset. It can be seen that the energy cost of different models decreases as the increasing of the batch size. When the batch size stays fixed, the energy cost is mainly related to the complexity of the model. However, the energy cost of all these models using our inference acceleration method is lower than the energy cost utilized by the TensorRT framework, especially for VGG19* and ResNet101*, the energy-saving effect is generally more than 50% by our proposed method. Table 10 illustrates the energy cost of the VGG19*, ViT-Tiny*, GoogLeNet*, and ResNet34* when performing the corresponding inference tasks on the testset of the Mini-ImageNet dataset. In Table 10, the same situation occurs that the energy cost of all these models by our inference acceleration method are lower than the energy cost utilized by the TensorRT framework. Therefore, the algorithm proposed in this article is more suitable for embedded devices.

4.8 GPU Utilization Test Under Different Batch Sizes

To further validate the scalability of our proposed solution in resource and workload dimensions, we conduct an in-depth study on a single model across varying batch sizes. The inference process is performed under the proposed method and the TensorRT's default method, respectively. The ViT-Tiny* model is conducted on the CIFAR-100 testset with different batch sizes of 1, 16, 32, and 64. As shown in Figure 10, for the inference process of ViT-Tiny*, the proposed method (a, c, e, g) demonstrates shorter inference times and higher GPU utilization in all tests. Different batch sizes simulate different concurrent processing sizes, either in high throughput high load or small batch

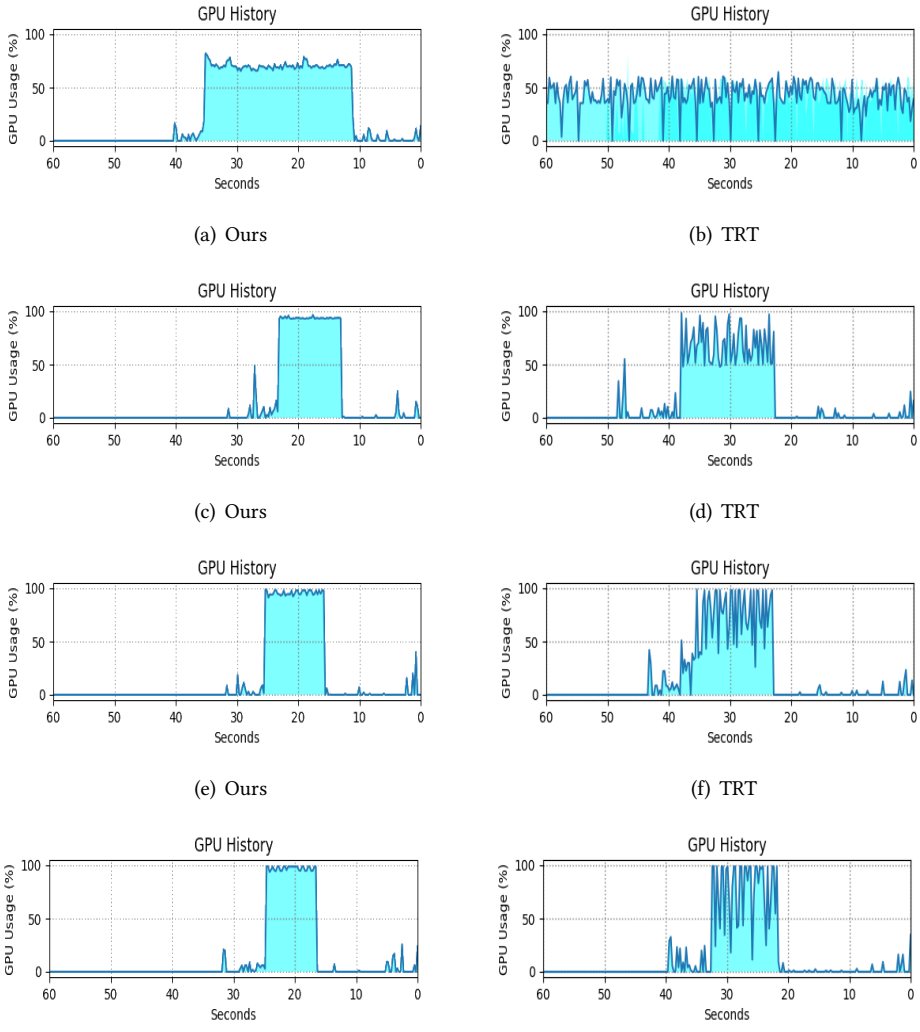
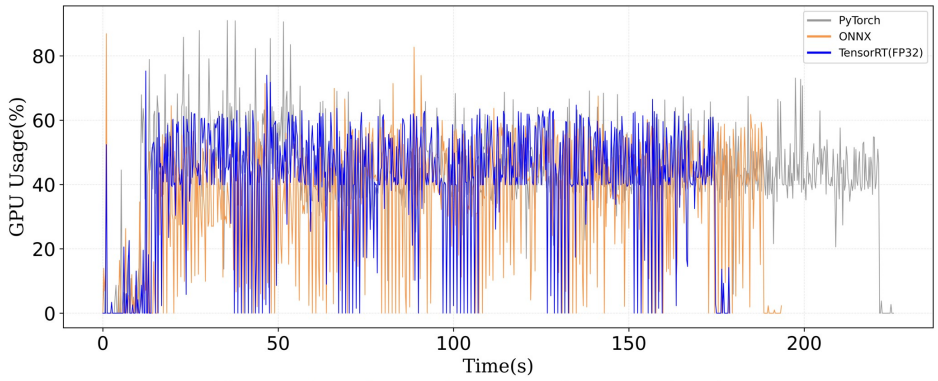


Fig. 10. The GPU utilization of the quantified ViT-Tiny model when performing model inference using the proposed method and the TensorRT's default method on the CIFAR-100 testset, respectively. The batch size in (a) and (b) is 1. The batch size in (c) and (d) is 16. The batch size in (e) and (f) is 32. The batch size in (g) and (h) is 64. TRT refers to the default scheduling method in TensorRT.

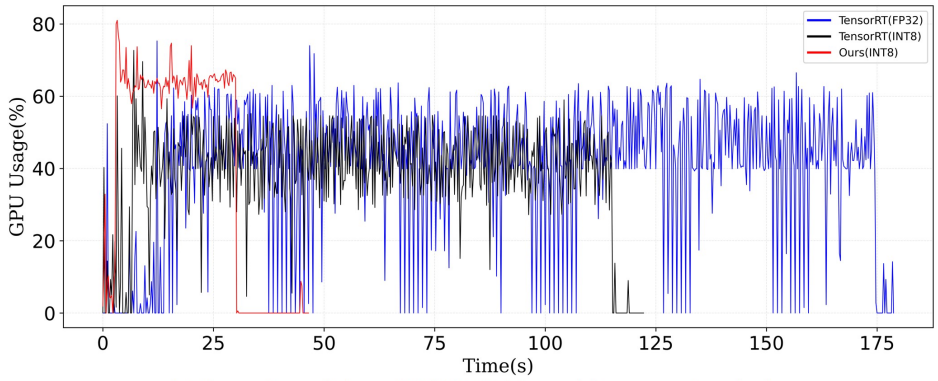
real-time environments, the proposed method is significantly effective while maintaining good stability. This shows that the proposed method is of great practical value.

4.9 Comparsion with Other Frameworks

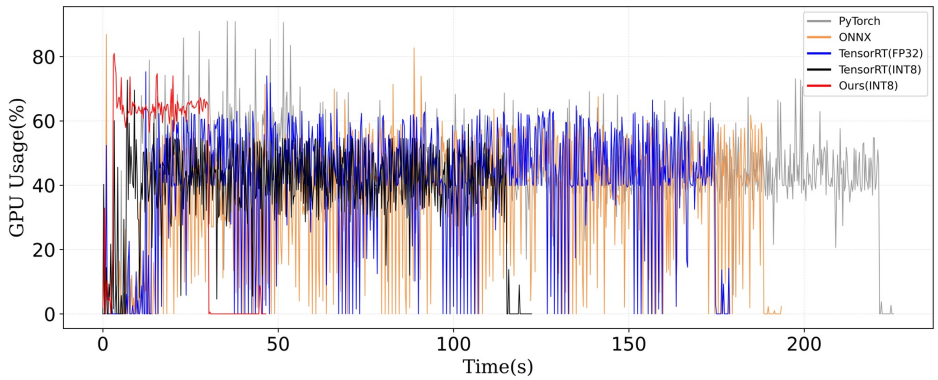
In order to demonstrate the substantive research significance and value of our work, we perform inference of the ViT-Tiny model on the CIFAR-100 dataset with batch sizes of 1. The inference experiments are conducted on PyTorch, ONNX, TensorRT and the comparison details included:



(a) Comparison of PyTorch, ONNX and TensorRT (FP32).



(b) Comparison of TensorRT (FP32/INT8), and the proposed method.



(c) Comparison of five frameworks.

Fig. 11. GPU usage of ViT-Tiny during inference on CIFAR-100 (batch size=1) with: PyTorch default, ONNX Runtime, TensorRT (FP32/INT8), and the proposed method.

PyTorch default inference, ONNX runtime inference, TensorRT engine without quantization (FP32), TensorRT engine with INT8 quantization, and our proposed thread-level stream scheduling method.

As shown in Figure 11, the results (for batch size=1) are presented with the horizontal axis representing inference time (s) and the vertical axis showing the measured GPU utilization (%). To

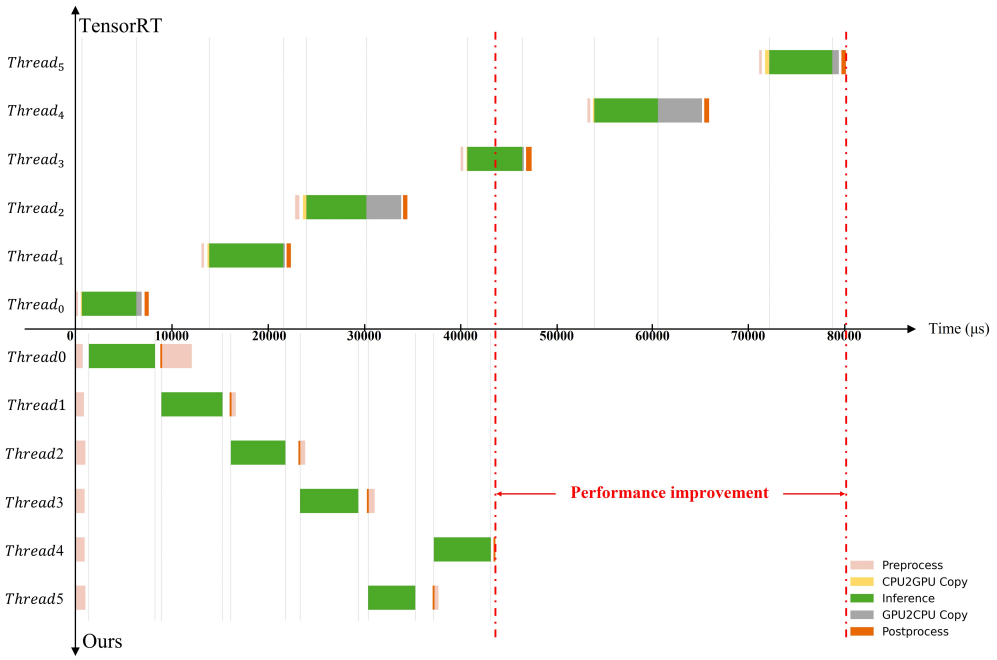


Fig. 12. Work flow comparison of TensorRT default scheduling method and the proposed method.

clearly illustrate the acceleration performance of our method, we subdivided the results into three figures, highlighting both the superiority of the TensorRT inference engine and the significant improvements achieved by our approach.

Figure 11 (a) shows that for the original DNN model, compared with PyTorch and ONNX, TensorRT delivers the best performance in GPU Usage. However, this advantage is relatively slight. Figure 11 (b) shows that although TensorRT with INT8 quantization achieves better acceleration performance, the GPU utilization rate is significantly reduced, which is an important starting point for our research. After applying our method, the GPU utilization rate significantly exceeds that of TensorRT using INT8 quantization alone and the inference time is greatly reduced. Figure 11 (c) summarizes the results of all comparative experiments, providing an intuitive illustration of the performance advantages of our approach.

4.10 Workflow Visualization of DNN's Inference Tasks

To verify the theoretical advantages of the proposed scheduling method in terms of workflow (compared with TensorRT's workflow), we conduct the visualization experiments on the workflow of actual inference tasks. The experiment was conducted on the ViT-tiny model with batch size of 1 on the CIFAR-100 test dataset. The proposed scheduling method and TensorRT default scheduling method are used to perform the same inference task of six batches, respectively. The execution time and sequence of all steps includes Preprocess, GPU2CPU Copy, Inference, GPU2CPU Copy, and Postprocess. The results of lantency test were shown as time spans in Figure 12.

To complete the same six batches, the proposed scheduling method only requires about 40000 us, while the TensorRT's default scheduling method takes about 80000 us. The result clearly validates the advantages of our work. Compared with the theoretical analysis mentioned in the article, the workflow of actual inference tasks is highly consistent with our theoretical analysis. For the

proposed scheduling method, the consistency is reflected in: (1) Redundant data copying is avoided by using managed memory management strategy. In the visualization experiment results, the proposed scheduling method did not have yellow and gray time spans, indicating the absence of GPU2CPU Copy and GPU2CPU Copy steps; (2) Thread-level scheduling can overlap the working time of CPU and GPU, avoiding GPU long waiting times. From the visualization results, the green time spans and pink/red time spans in the proposed scheduling method can appear in the same time period, and the time interval between adjacent green time spans is much smaller than the TensorRT default scheduling method. In summary, the results of visualization fully demonstrate that by implementing the theoretical method proposed in this article, the expected advantages have been achieved, providing an efficient solution for model inference on NX embedded devices.

5 Conclusion

In this article, we presented a novel thread-level CUDA stream scheduling method based on unified memory to improve GPU utilization during DNNs' real-time inference. Firstly, we analyzed the process of model inference on an integrated device, which demonstrates the heterogeneous tasks during the inference process of DNNs' model. Then, we compared different GPU memory management strategies and adopted the MMS to implement unified memory management between heterogeneous processors. Finally, based on unified memory management, a heterogeneous pipeline-style task scheduling method was designed to improve the utilization of on-chip resources, which adopts a thread-level task scheduling algorithm to maximize the GPU throughput for various DNN models. Finally, extensive experiments were conducted on the Jetson Xavier NX platform, which demonstrates the effectiveness of the proposed method.

References

[1] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2019. High-throughput cnn inference on embedded arm big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2254–2267.

[2] Soroush Bateni, Zhendong Wang, Yuankun Zhu, Yang Hu, and Cong Liu. 2020. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 310–323.

[3] Yingfeng Cai, Tianyu Luan, Hongbo Gao, Hai Wang, Long Chen, Yicheng Li, Miguel Angel Sotelo, and Zhixiong Li. 2021. YOLOv4-5D: An effective and efficient object detector for autonomous driving. *IEEE Transactions on Instrumentation and Measurement* 70 (2021), 1–13.

[4] Changyu Chen, Guangchao Geng, Heyang Yu, Zixuan Liu, and Quanyuan Jiang. 2021. An end-cloud collaborated framework for transferable non-intrusive load monitoring. *IEEE Transactions on Cloud Computing* 11, 2 (2021), 1157–1169.

[5] Krishna Teja Chitty-Venkata, Sparsh Mittal, Murali Emani, Venkatram Vishwanath, and Arun K Somani. 2023. A survey of techniques for optimizing transformer inference. *Journal of Systems Architecture* 144 (2023), 250–256.

[6] Jake Choi, Hojun You, Chongam Kim, Heon Young Yeom, and Yoonhee Kim. 2021. Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on tegra SoC. *Concurrency and Computation: Practice and Experience* 33, 4 (2021), e6018.

[7] Mohammad Dashti and Alexandra Fedorova. 2017. Analyzing memory management methods on integrated CPU-GPU systems. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 59–69.

[8] Zhentao Fan, Xianhao Wu, Xiang Chen, and Yufeng Li. 2023. Learning to see in nighttime driving scenes with inter-frequency priors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4218–4225.

[9] Xingyu Feng, Lingjie Li, Tenglong Wang, Weitao Xu, Jin Zhang, Bo Wei, and Chengwen Luo. 2023. CoBC: A blockchain-based collaborative inference system for internet of things. *IEEE Internet of Things Journal* 10, 24 (2023), 21389–21400.

[10] Yakun Huang, Xiuquan Qiao, Schahram Dustdar, Jianwei Zhang, and Jiulin Li. 2022. Toward decentralized and collaborative deep learning inference for intelligent iot devices. *IEEE Network* 36, 1 (2022), 59–68.

[11] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: Efficient CPU-GPU co-execution for deep learning inference on mobile devices.. In *MobiSys*, Vol. 22. 209–221.

[12] Shuiwang Li, Yangxiang Yang, Dan Zeng, and Xucheng Wang. 2023. Adaptive and background-aware vision transformer for real-time UAV tracking. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 13989–14000.

[13] Zhuo Li, Hengyi Li, and Lin Meng. 2023. Model compression for deep neural networks: A survey. *Computers* 12, 3 (2023), 60.
[14] NVIDIA. 2024. *Jetson Modules*. Retrieved from https://developer.nvidia.com/embedded/jetson-modules
[15] NVIDIA. 2024. *Jetson Nano Developer Kit*. Retrieved from https://developer.nvidia.com/embedded/jetson-nano-developer-kit
[16] NVIDIA. 2024. *Jetson Xavier NX Developer Kit*. Retrieved from https://developer.nvidia.com/embedded/jetson-xavier-nx-devkit
[17] Dheeraj Peri, Jhalak Patel, and Josh Park. 2020. Deploying quantization-aware trained networks using TensorRT. In *GPU Technology Conference*.
[18] Tanzila Saba, Amjad Rehman, Khalid Haseeb, Teg Alam, and Gwanggil Jeon. 2023. Cloud-edge load balancing distributed protocol for IoE services using swarm intelligence. *Cluster Computing* 26, 5 (2023), 2921–2931.
[19] Amna Shahid and Malaika Mushtaq. 2020. A survey comparing specialized hardware and evolution in TPUs for neural networks. In *2020 IEEE 23rd International Multitopic Conference (INMIC)*. IEEE, 1–6.
[20] Nir Shlezinger and Ivan V Bajić. 2022. Collaborative inference for AI-empowered IoT devices. *IEEE Internet of Things Magazine* 5, 4 (2022), 92–98.
[21] Philipp Terhörst, Malte Ihlefeld, Marco Huber, Naser Damer, Florian Kirchbuchner, Kiran Raja, and Arjan Kuijper. 2023. Qmagface: Simple and accurate quality-aware face recognition. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 3484–3494.
[22] Delia Velasco-Montero, Bart Goossens, Jorge Fernández-Berni, Ángel Rodríguez-Vázquez, and Wilfried Philips. 2023. A pipelining-based heterogeneous scheduling and energy-throughput optimization scheme for CNNs leveraging apache TVM. *IEEE Access* 11 (2023), 35007–35021.
[23] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. 2020. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 46–49.
[24] Zhiyuan Xu, Dejun Yang, Chengxiang Yin, Jian Tang, Yanzhi Wang, and Guoliang Xue. 2021. A co-scheduling framework for DNN models on mobile and edge devices with heterogeneous hardware. *IEEE Transactions on Mobile Computing* 22, 3 (2021), 1275–1288.
[25] Dingxi Yang, Zhijin Qin, Liting Wang, Xiaoming Tao, Fang Cui, and Hengjiang Wang. 2023. An end-cloud computing enabled surveillance video transmission system. In *2023 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 209–214.
[26] Qunsong Zeng, Yuqing Du, Kaibin Huang, and Kin K Leung. 2021. Energy-efficient resource management for federated edge learning with CPU-GPU heterogeneous computing. *IEEE Transactions on Wireless Communications* 20, 12 (2021), 7947–7962.
[27] Weiting Zhang, Dong Yang, Haixia Peng, Wen Wu, Wei Quan, Hongke Zhang, and Xuemin Shen. 2021. Deep reinforcement learning based resource management for DNN inference in industrial IoT. *IEEE Transactions on Vehicular Technology* 70, 8 (2021), 7605–7618.
[28] Yuxiao Zhou and Kecheng Yang. 2022. Exploring tensorrt to improve real-time inference for deep learning. 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)*. IEEE, 2011–2018.

Received 21 November 2024; revised 1 September 2025; accepted 4 October 2025