BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference

Neiwen Ling
The Chinese University of Hong Kong
Hong Kong, China
lingnw@link.cuhk.edu.hk

Xuan Huang
The Chinese University of Hong Kong
Hong Kong, China
1155136647@link.cuhk.edu.hk

Zhihe Zhao
The Chinese University of Hong Kong
Hong Kong, China
1155170475@link.cuhk.edu.hk

Nan Guan
City University of Hong Kong
Hong Kong, China
nanguan@cityu.edu.hk

Zhenyu Yan
The Chinese University of Hong Kong
Hong Kong, China
zyyan@cuhk.edu.hk

Guoliang Xing[†]
The Chinese University of Hong Kong
Hong Kong, China
glxing@cuhk.edu.hk

ABSTRACT

In recent years, Deep Neural Network (DNN) has been increasingly adopted by a wide range of time-critical applications running on edge platforms with heterogeneous multiprocessors. To meet the stringent timing requirements of these applications, heterogeneous CPU and GPU resources must be efficiently utilized for the inference of multiple DNN models. Such a cross-processor real-time DNN inference paradigm poses major challenges due to the inherent performance imbalance among different processors and the lack of real-time support for cross-processor inference from existing deep learning frameworks. In this work, we propose a new system named BlastNet that exploits *duo-block - a new model inference abstraction* to support highly efficient cross-processor real-time DNN inference. Each duo-block has a dual model structure, enabling efficient fine-grained inference alternatively across different processors. BlastNet employs a novel block-level Neural Architecture Search (NAS) technique to generate duo-blocks, which accounts for computing characteristics and communication overhead. The duo-blocks are optimized at design time and then dynamically scheduled to achieve high resource utilization of heterogeneous CPU and GPU at runtime. BlastNet is implemented on an indoor autonomous driving platform and three popular edge platforms. Extensive results show that BlastNet achieves 35.07 % less deadline missing rate with a mere 1.63% of model accuracy loss.

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; **Embedded systems**; • **Computing methodologies** → **Concurrent computing methodologies**; **Computer vision tasks**.

[†]Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SenSys '22, November 6–9, 2022, Boston, MA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9886-2/22/11...$15.00
https://doi.org/10.1145/3560905.3568520

KEYWORDS

Edge Artificial Intelligence, Multi-DNN Concurrent Execution, Real-Time Scheduling, CPU-GPU Heterogeneous Platform, Neural Architecture Search, On-device Deep Learning

ACM Reference Format:

Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing[†]. 2022. BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '22), November 6–9, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3560905.3568520

1 INTRODUCTION

CPU-GPU heterogeneous architectures are increasingly popular for edge platforms, thanks to their capability of accelerating DNN inference [33, 35, 37]. For example, NVIDIA Xavier [40], a mainstream industrial edge platform, has an 8-core CPU and a Volta GPU. Such heterogeneous multiprocessor platforms can enable a wide class of applications where multiple real-time DNN inference tasks are conducted on the edge concurrently, such as autonomous driving [28], smart roadside infrastructure [43], and embedded computer vision [57]. For example, a smart lamppost [60] equipped with an NVIDIA TX2 platform (a 6-core ARM CPU + a Pascal GPU) may run concurrent real-time DNN inference for license plate detection [56], pedestrian/vehicle tracking [7, 29], and even collision detection and warning for autonomous driving vehicles [22, 36].

However, existing works have not fully explored the heterogeneous processors on these platforms for DNN inference. First, most current solutions focus on accelerating DNN inference on a single AI accelerator (e.g., GPU or NPU) [8, 47]. Quantization techniques [9, 13] can achieve efficient DNN inference on CPU, while some search for efficient model architectures with direct measurements on each processor [2, 6, 49, 54]. However, these techniques cannot be extended directly to utilize the CPU-GPU heterogeneous architecture. Second, existing Deep Learning (DL) frameworks such as PyTorch and TensorFlow only support monolithic DNN model scheduling, which allocates the whole DNN model inference to either CPU or GPU statically. In addition, an urgent DNN task cannot preempt the inferences of other DNNs in time. Recent studies [55, 59] partition the DNN model into sub-tasks for fine-grained DNN inference scheduling. However, since the GPU is generally more powerful than the CPU, offloading to a spare process can still lead to non-trivial waiting time.

In this work, we propose a new Block-Level model optimizAtion and Scheduling sysTem - BlastNet, which supports highly efficient cross-processor real-time inference of concurrent DNNs on heterogeneous multiprocessor platforms. The design of BlastNet is based on *duo-block - a new model inference abstraction*. Each duo-block has a dual model structure, consisting of a CPU block and a GPU block, which hence enables dynamic alternative execution of DNN across processors. To optimize the DNN model for efficient inference on different processors, we propose a novel *duo-block generation* algorithm that jointly considers layer-level computing and communication characteristics, as well as the operator fusion rules. By adopting a neural architecture search algorithm, our approach generates highly optimized CPU/GPU blocks that speed up the execution on different processors. Lastly, we design a new *dynamic cross-processor scheduler* which prioritizes each DNN duo-block based on its urgency and supports flexible and dynamic scheduling of duo-blocks on CPU-GPU heterogeneous resources, which meets stringent timing requirements of concurrent real-time DNN applications.

We implement BlastNet on three mainstream CPU-GPU heterogeneous edge platforms and an indoor autonomous driving platform. Our extensive experiments show that BlastNet can enable cross-processor real-time DNN model inference to achieve satisfactory real-time performance. Moreover, compared to several state-of-the-art baselines with fine-/coarse-grained scheduling policies, BlastNet can reduce deadline missing rate up to 35.07% while only sacrificing negligible DNN model accuracy. In summary, this paper makes the following key contributions:

- We propose a novel model inference abstraction - duo-block to support efficient DNN model inference on CPU-GPU heterogeneous platforms.
- We design a duo-block generation algorithm that optimizes DNN models to different processors in the block-level granularity based on Neural Architecture Search (NAS) techniques.
- We design a dynamic cross-processor scheduler that supports flexible and dynamic allocation of model blocks on CPU-GPU heterogeneous resources at runtime, which significantly improves the CPU/GPU utilization for concurrent real-time DNN inferences.
- We implement BlastNet and evaluate the performance through extensive experiments on three mainstream CPU-GPU heterogeneous edge platforms. Our results provide key insights into supporting real-world time-critical DNN applications on heterogeneous edge platforms.

The rest of the paper is organized as follows. Section 2 reviews the related work, Section 3 introduces the background and a motivational case, Section 4 describes the system design of BlastNet, Section 5 illustrates the detailed component design of BlastNet, Section 6 evaluates the performance of BlastNet, Section 7 discusses the scalability of BlastNet and Section 8 concludes the paper.

2 RELATED WORK

Model Partitioning for Cross-Processor DNN Inference. Considering the limited resource on edge platforms, some approaches offload the workloads of DNN inference to the cloud [26, 61–63]. A common solution is partitioning the DNN model and offloading partial model inference to the cloud. This idea can be extended to heterogeneous platforms for efficient resource utilization [11, 19, 20, 55, 58, 59]. DART [55] partitions a DNN model in a layer-level granularity and schedules layers among CPU and GPU processors. Heimdall [59] proposes to partition a DNN model into the unit of consecutive DNN operators, which are scheduled efficiently to share the GPU and also offloaded some units to the CPU. Band [19] partitions a DNN into a set of subgraphs, and dynamically selects a schedule of subgraphs from multiple possible schedules at run-time.

However, obtaining significant real-time performance gains is challenging due to the large differences between different processors. Offloading a single DNN layer from a powerful processor to a weaker processor may not accelerate the execution, but instead can block the DNN inference due to slower execution on the weaker processor. We will show a more detailed analysis of the challenges arising from this DNN inference diversity on different processors in Section 3.

Hardware-Aware NAS. Hardware-aware neural architecture search (NAS) aims to find effective neural network architectures for specific platforms automatically. It has recently been applied to meet various performance requirements on resource-constrained hardware platforms [2, 6, 49, 54]. ProxylessNAS [2] searches for neural architectures for a specific processor (e.g., CPU and GPU) with performance metrics on different hardware platforms. MnasNet [49] performs hardware-aware NAS via directly measuring real-world inference latency on smartphones. However, these works search the entire DNN model architecture space for a specific processor, which cannot be applied efficiently to heterogeneous platforms with different types of processors.

DNN Model Replacement. Another research direction related to our work is DNN Model Replacement. DFN [27] selects a set of well-known functions to replace the DNN model. However, DFN focuses on replacing the entire DNN model, which cannot be directly generalized for DNN model adaptation in the layer-level granularity. Another work considering layer-wise replacement of DNN models is LegoDNN [12], which enables block-level DNN training and scaling. However, LegoDNN is designed for platforms with a single type of processor. Moreover, it does not account for the (possibly high) communication overhead between layers for cross-processor DNN inference.

Joint Model Optimization and Scheduling. LalaRAND [21] couples the model quantization technique with fine-grained CPU/GPU resource allocation to address the asymmetric nature of DNN inference on CPU-GPU platforms. However, the benefits from quantization are hardware dependent, with many factors affecting the quantization speed up [9]. Besides, layer-level scheduling may lead to high overhead for DNN inference due to communication overhead. Asymo [51] adopts matrix multiplication partitioning and task scheduling to alleviate the unbalanced task distribution on mobile CPUs. Both works focus on optimizing DNN inference on specific processors and do not apply to other heterogeneous platforms with different types of processors. RT-mDL [33] supports real-time DL tasks via joint model scaling and scheduling, which is applicable to

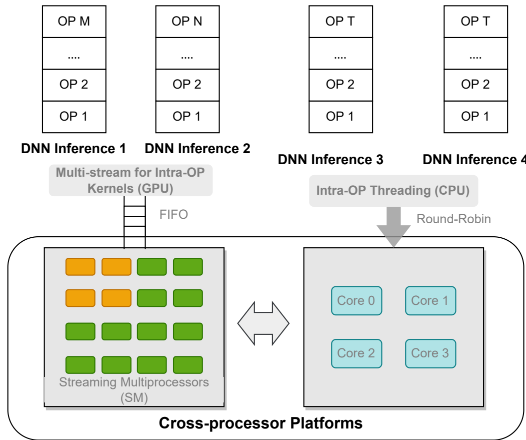


Figure 1: Concurrent DNN inference under PyTorch framework on heterogeneous CPU-GPU platforms.

heterogeneous platforms with different types of processors. However, RT-mDL does not optimize the model architecture for different processors and adopts the fixed-priority model-level DNN scheduling at runtime, which cannot utilize cross-processor resources efficiently, as shown in our results (Section 3.2 and 6.4). Compared with the above approaches, our work provides a new approach that exploits holistic block-level model architecture optimization and scheduling for real-time cross-processor DNN reference.

3 BACKGROUND AND MOTIVATION

In this section, we first discuss the status quo for DNN inference on heterogeneous CPU-GPU platforms, and then profile the compute characteristics of cross-processor DNN inference. The results provide key insights into our design of duo-block. In our profiling experiments, we select three DNN models - AlexNet [25], VGG11 [46], ResNet18 [15], which have been adopted by a wide range of edge applications. The training of these models is based on the CIFAR-10 [24] (for image classification) and GTSRB [48] (for sign recognition) datasets, and the inference is executed and profiled on a typical desktop-class platform (Intel i9 CPU + NVIDIA RTX 2080 GPU) and a GPU-accelerated edge platform (NVIDIA AGX Xavier).

3.1 DNN Inference on CPU-GPU Platform

In this work, we focus on heterogeneous architectures that consist of CPU and GPU processors, which are increasingly adopted by embedded and edge systems to accelerate the inference of DNN. For example, Google Pixel 6 [10] has an 8-core CPU as well as a Mali-G78 MP20 GPU. NVIDIA Xavier [40], an industrial edge platform, has an 8-core CPU and a Volta GPU.

Current mainstream deep learning frameworks, such as PyTorch, TensorFlow, and MXNet, monolithically allocate heterogeneous resources for concurrently executed DNN models. For example, as shown in Fig. 1, PyTorch allocates each DNN model to a single type of processor (e.g., CPU or GPU) before its execution. One or more inference threads then execute the model's forward pass on the specified processor. Each thread invokes a JIT interpreter that executes the operators (OPs) of a model inline, one by one

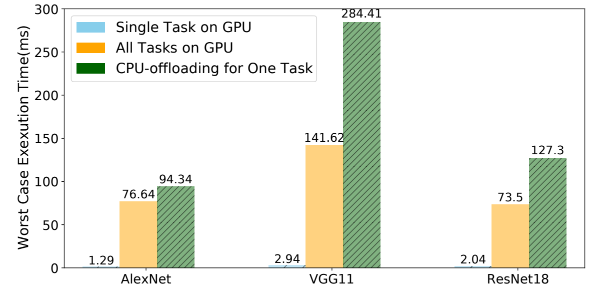


Figure 2: Worst-case execution time for concurrent DNN model inference on CPU-GPU platform under different resource allocation strategies.

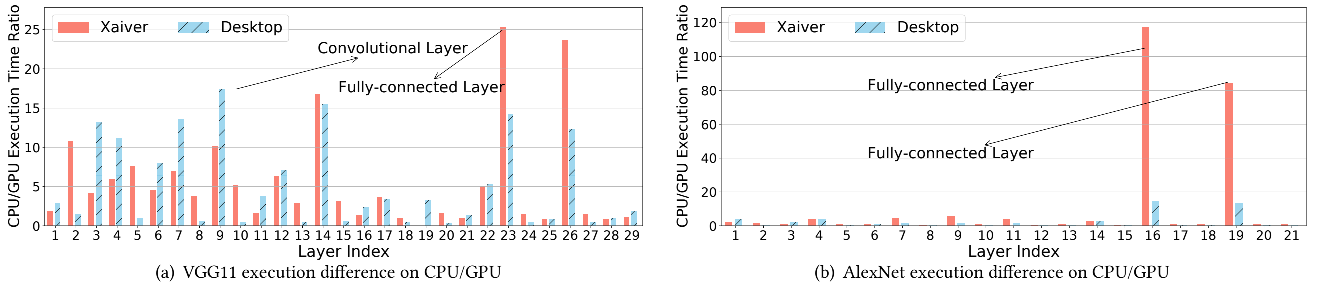
[42]. In such a case, urgent tasks assigned to an occupied processor can easily miss their deadlines, even though there exists resource available on other processors. Since the processor affinity of each model is fixed before its execution, the DNN model inference cannot be offloaded to different processors at runtime.

Moreover, as shown in Fig. 1, DNN models allocated to GPU launch intra-OP kernels (e.g., aten::cudnn_convolution) to the stream of GPU. A sequence of kernels in the stream is then executed in FIFO order [39]. Similarly, DNN models allocated to CPU execute their OPs in a round-robin manner (possibly on different cores). In both cases, the DNN models that occupy the same processor (either CPU or GPU) must contend for the resource, in spite of their real-time requirements.

3.2 Model-Level DNN Inference

To support concurrent model inference on CPU-GPU heterogeneous platforms, the common wisdom is to allocate models to different processors for resource sharing. In this section, we investigate whether such a model-level strategy enables efficient concurrent DNN inference. To this end, we profile the model-level compute characteristics in the presence of resource contention from concurrent DNN inference. As shown in Fig. 2, we profile the worst-case execution time (WCET) of three types of DNN models under different resource allocation strategies. WCET is a typical performance metric for understanding the timing behavior of real-time tasks under the worst case (e.g., with the most severe resource contention) [53]. We run a total of 10 DNN models of each type concurrently under two different resource allocation strategies, i.e., running all tasks on GPU or running all but one task on CPU (i.e., offloading one of the tasks to CPU). We note that offloading more tasks is not beneficial due to the inefficiency of model inference on the CPU. As a comparison, we also profile the WCET of running a single DNN inference on GPU.

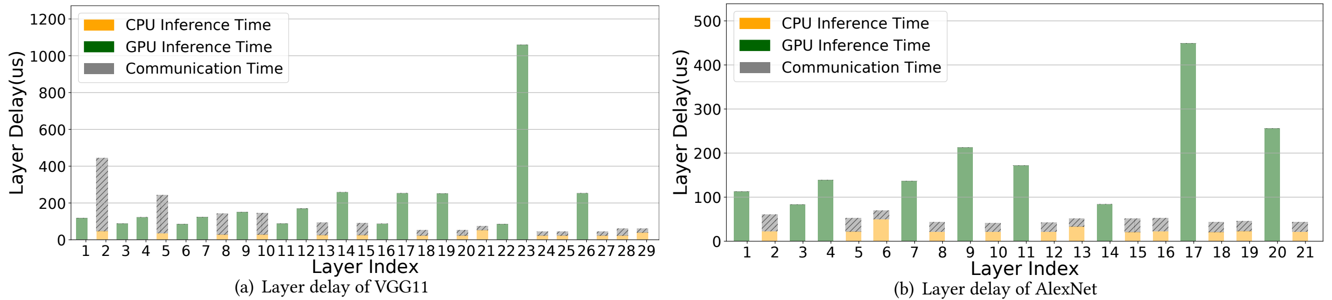
Specifically, we create an independent thread for each DNN model inference and allocate it to the CPU or GPU on a typical desktop-class platform. We run each DNN inference task 1,000 times and record the WCET. To avoid the biases caused by long model initialization delays, we exclude the first five runs in the WCET calculation. Fig. 2 clearly demonstrates the inefficient model execution under both resource allocation strategies. For example, the WCETs of running all 10 inference tasks on GPU are 60.97×, 48.08×, and 36.06× the delays of running the single model inference, for AlexNet, VGG11, ResNet18, respectively. Such a significant



(a) VGG11 execution difference on CPU/GPU

(b) AlexNet execution difference on CPU/GPU

Figure 3: CPU/GPU execution ratio for each DNN layer.



(a) Layer delay of VGG11

(b) Layer delay of AlexNet

Figure 4: Layer delay by executing each layer on the processor with the shortest inference time (CPU Utilization: 9.05% **for VGG11,** 13.37% **for AlexNet, GPU Utilization:** 65.59% **for VGG11,** 73.39% **for AlexNet, Communication Overhead:** 25.35% **for VGG11,** 13.37% **for AlexNet).**

delay increase for concurrent inference is caused by the resource contention under the model-level execution mechanism, which validates the observation discussed in Section 3.1. Moreover, it is shown that offloading tasks to the CPU is not helpful, and suffers even higher delays. This is because the model inference on the CPU is much slower than on the GPU. In summary, **allocating DNN models to CPU and GPU in a model-level granularity leads to severe resource contention.**

3.3 Layer-level DNN Inference

A DNN model usually consists of various layers such as convolutional, fully-connected, and max pooling layers. Thus, an idea to achieve better resource utilization than the model-level allocation strategy is executing DNN layers on different processors. In this section, we examine the performance imbalance of DNN inference among processors and then investigate whether the layer-level allocation strategy is efficient on CPU-GPU platforms. Since the current DL frameworks such as PyTorch and TensorFlow do not support layer-level resource allocation, we execute the entire model and profile the inference and communication overhead of each layer, instead of executing each layer separately.

We measure the execution time of each layer of VGG11 on CPU and GPU respectively of a typical desktop-class platform and calculate the CPU/GPU execution ratio of each layer. We fix the CPU affinity to one core to avoid switch overhead among different CPU cores. Fig. 3 shows the execution time ratio of CPU and GPU of each layer. The results show that different layers incur significantly unbalanced workloads on CPU and GPU. For example, as shown in Fig. 3(a), the first fully-connected layer of VGG11 runs 25.3× faster on GPU than CPU, while the second convolutional layer of VGG11 is only 5.9× faster. This is because some DNN layers such as convolutional and fully-connected layers contain massive parallel computing operations that are better optimized for execution on GPU than CPU. We also verify whether the results also hold for other computing platforms. We perform a similar experiment using NVIDIA AGX Xavier and AlexNet, which is another typical DNN model. The results show a similar conclusion, demonstrating that such a layer-wise diverse performance imbalance between CPU and GPU exists in mainstream edge platforms and DNN models. **Such kind of imbalance may become a barrier for cross-processor real-time DNN inference**, because a task with high priority may be blocked by its own execution on a weaker processor.

Based on the profiled latency of each layer on both CPU and GPU, we then choose the more efficient processor as the one that the layer would be allocated on. We note that we use this naive layer-level allocation strategy to understand the compute characteristics of different layers during model inference, while the state-of-the-art layer-level CPU/GPU scheduling approaches are discussed in Section 2 and compared as baselines in our experiments in Section 6. Fig. 4 shows the inference and communication overhead of each layer. We observe that, compared with the model-level strategy in Section 3.3, the CPU is now utilized for layer inference. However, the workload between CPU and GPU is still highly unbalanced. In the case of VGG11, the total execution time on CPU is 440.7 μs, which is only about 13% of that of GPU. In other words, the CPU remains idle most of the time while the GPU is usually occupied. More importantly, the communication between CPU and GPU costs 1234.5 μs, which accounts for 25.35% of the total execution time. The results indicate that **allocating DNN models to CPU and**

GPU in a layer-level granularity may cause low resource utilization and significant layer switching overhead.

3.4 Summary

Our key observation in this section is that, when multiple DNN models are executed concurrently, allocating them to CPU/GPU at either model- or layer-level granularity cannot utilize resources efficiently. The model-level allocation strategy often causes severe resource contention on the GPU while leaving the CPU idle. Although the layer-level allocation improves resource utilization by offloading some layers to the CPU, it may lead to frequent layer switching and significant communication overhead. Moreover, offloading a single DNN layer from a powerful processor to a weaker processor may lead to the blocking of DNN inference due to the vast performance imbalance between processors. This kind of imbalance is highly diverse among different DNN layers. These results motivate us to find a new abstraction of model partition and scheduling to achieve better real-time concurrent DNN inference on heterogeneous CPU-GPU platforms.

4 PROBLEM DEFINITION AND SYSTEM OVERVIEW

4.1 Problem Definition

In this work, we consider the problem of supporting concurrent real-time DNN inference on CPU-GPU heterogeneous platforms. The key idea of BlastNet is to optimize the architecture of DNN models in block-level granularity at design time, and support flexible and dynamic allocation of optimized model blocks on CPU-GPU resources at runtime. Specifically, we define the original model for each DL task τ_i as $M = \{L_0, L_1, ..., L_m\}$, where L_m denotes the mth layer of the DNN model. We aim to construct a model with duo-blocks (B_k^C, B_k^G) (C refers to CPU, and G refers to GPU), which is defined as $M^{duo} = \{(B_0^C, B_0^G), (B_1^C, B_1^G), ..., (B_k^C, B_k^G)\}$. Each duo-block consists of a CPU block B_k^C and a GPU block B_k^G. Each CPU/GPU block consists of multiple adjacent model layers highly optimized for either GPU or CPU. In order to meet real-time requirements for each DNN inference, we aim to schedule the execution of the GPU block B_k^G or the CPU block B_k^C of a duo-block based on run-time resource usage and task urgency dynamically. For example, the execution path can be $B_0^G \rightarrow B_1^C \rightarrow B_2^C, ..., \rightarrow B_k^G$ or $B_0^C \rightarrow B_1^C \rightarrow B_2^G, ..., \rightarrow B_k^G$. Overall, we aim to maximize the real-time performance of all DL inferences (i.e., minimize the total deadline missing rates), while meeting a given accuracy bound on each DNN model, i.e., $ACC(M^{exe}) > Acc_Bound$. The above problem formulation can also be extended to achieve differentiated levels of accuracy among different DL tasks by setting a user-specified bound $> Acc_Bound_i$ for the accuracy of each DL task.

4.2 System Architecture

Fig. 5 shows a bird-eye view of BlastNet. BlastNet consists of two major components, i.e., duo-block generation and dynamic cross-processor scheduling, collaborating to exploit duo-blocks for cross-processor real-time DNN model inference. A key novelty of BlastNet is that each duo-block has a dual model structure, consisting of a CPU block and a GPU block highly optimized for CPU and GPU,

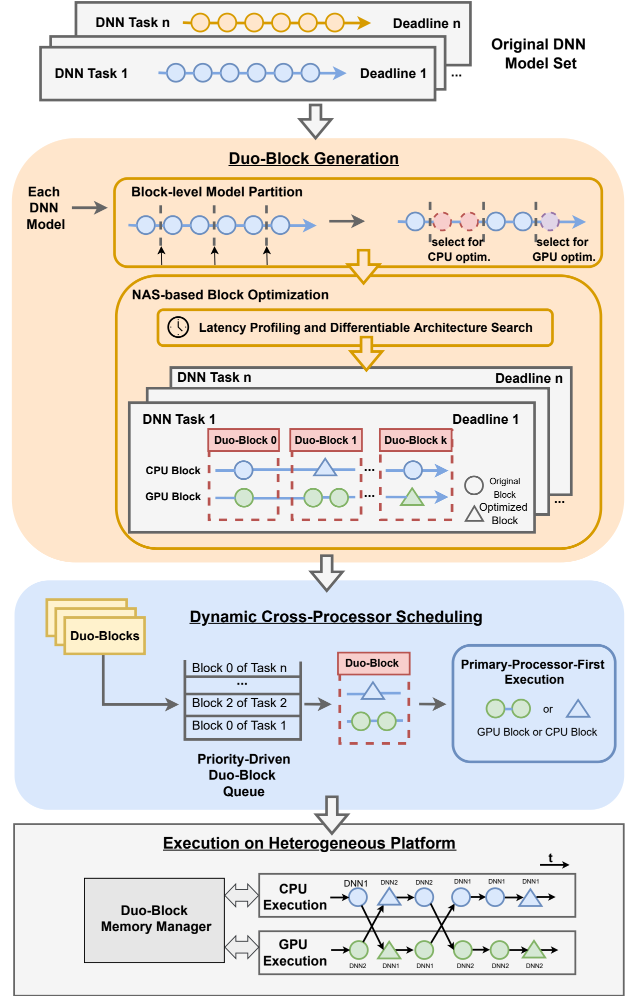


Figure 5: System architecture of BlastNet

respectively. Such design of duo-block enables dynamic alternative execution of DNN across processors, which provides more scheduling flexibility for concurrent DNN execution.

BlastNet generates duo-blocks via offline block-level DNN partition and optimization. We first partition the DNN models into blocks by jointly considering the layer-level computing and communication characteristics as well as the operator fusion rules. Then, each block is optimized for execution on CPU and GPU, constructing a *duo-block* structure to support runtime dynamic scheduling. A key challenge of duo-block generation is that optimizing two versions of each block for GPU and CPU respectively may incur a vast computation and storage overhead. We address this challenge by optimizing only the subset of blocks that 1) account for a large portion of the overall inference time, and 2) exhibit highly imbalanced inference performance on CPU and GPU, which presents more opportunities for leveraging the heterogeneous processors. Specifically, for each block to be optimized, we search for an optimal model architecture via a two-stage Neural Architecture Search (NAS)-based block optimization approach. First, we optimize the

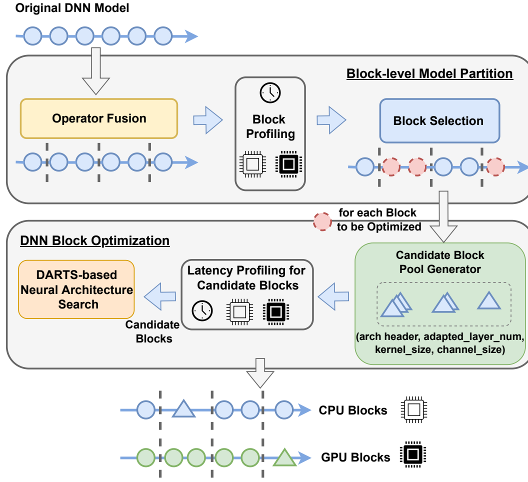
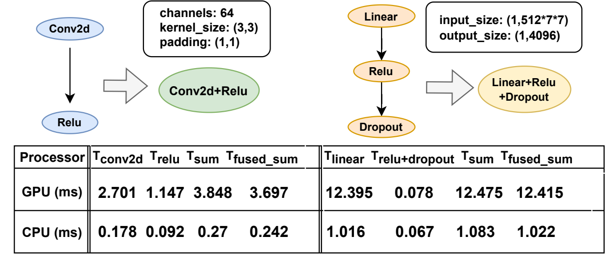


Figure 6: Generation procedure for cross-processor block



Processor	T_{conv2d}	T_{relu}	T_{sum}	T_{fused_sum}	T_{linear}	$T_{relu+dropout}$	T_{sum}	T_{fused_sum}
GPU (ms)	2.701	1.147	3.848	3.697	12.395	0.078	12.475	12.415
CPU (ms)	0.178	0.092	0.27	0.242	1.016	0.067	1.083	1.022

Figure 7: Operator fusion and its benefit (evaluated under torchscript on the desktop platform with NVIDIA RTX 2060 GPU). $T_{operator_name}$ denotes the execution time of the operator. T_{sum} denotes the sum of T_{conv2d}, T_{relu} or T_{linear}, $T_{dropout}$. T_{fused_sum} denotes the execution time of fused operators.

search space to meet the latency constraints based on the latency profiling on the target processor (i.e., CPU or GPU). Second, we aim to find the specialized network architecture in the optimized search space via the Differentiable Architecture Search (DARTS) algorithm [34].

At runtime, BlastNet adopts a novel dynamic cross-processor scheduling mechanism that schedules each DNN block on CPU or GPU based on its urgency and current system resources. Specifically, in order to meet the real-time requirements of each DNN inference task, BlastNet first orders the execution of each duo-block based on its task urgency in a priority-driven duo-block queue. A challenge in the design of BlastNet is how to maintain the bounded accuracy since the CPU and GPU blocks of the same duo-block may yield highly different levels of accuracy. BlastNet addresses this problem by tracking the execution path of each model and accounting for the dynamic accuracy loss in a new primary-processor-first execution mechanism. In this way, BlastNet can trade accuracy for latency flexibly according to the utilization of each processor at runtime. Lastly, BlastNet minimizes the cross-processor communication overhead by avoiding the data copying between blocks that execute on the same processor.

5 DESIGN OF BLASTNET

5.1 Cross-processor Duo-block Generation

Fig. 6 shows the design of our cross-processor duo-block generation process. In order to optimize the DNN model to different processors for more efficient execution, we design a cross-processor block generation algorithm, where we partition DNN models into blocks and optimize them to enable more efficient execution on GPU or CPU, respectively.

The main challenge here is how to achieve the fine-grained DNN model partition while still allowing the opportunity for joint architecture optimization between adjacent layers. To address this challenge, we design a block-level model partition mechanism that fuses DNN layers into blocks based on operator fusion rules and layer characteristics, and then merges adjacent blocks that need to be optimized. However, each duo-block contains two optimized versions of the same block for both CPU and GPU, which may incur a vast computation and storage overhead. To address this issue, we carefully identify a subset of blocks for optimization based on latency profiling on the target platform. Next, for each DNN block that needs to be optimized, we search for a more efficient block architecture for CPU/GPU execution, which executes faster on the target processor, while having a similar accuracy. A common heuristic to reduce the inference time of DNN is reducing the number of computation operations via model compression techniques such as filter pruning or quantization. However, as shown in previous work [50], only reducing computation operations or memory accesses do not always lead to lower inference latency. It is critical to consider the computing characteristic of DNN operators on the target platform. Hence, we propose to customize processor-friendly candidate blocks for the target processor, and then search for efficient blocks via NAS techniques.

5.1.1 Block-level Model Partition. We first fuse DNN layers into blocks based on the general operator fusion rules. Current DL framework such as PyTorch and TensorFlow optimizes the DNN model inference by optimization techniques such as operator fusion, which combines multiple DNN operators into a single kernel instead of storing the intermediate results in on-board memory. Operator fusion eliminates unnecessary transmission of intermediate computing results, and thus reduces the overhead of launch and synchronization. Figure 7 demonstrates two examples of operator fusion on two operator nodes and the benefit. The operator fusion rule we considered in this work is the general platform-independent graph optimization in TVM [4] (*tvm.relay.transform.FuseOps (fuse_opt_level)*), which is applicable for cross-processor execution and widely used in many DL frameworks. We fuse two blocks when transmitting a block's results between processors takes a longer time than executing it on the faster processor.

As we discussed in Section 3.3, layers in the same DNN model show different execution performances on different processors. We first measure the inference time of each block on all processors in the target edge platform (i.e., CPU or GPU) to determine the primary and the secondary processors for each block. The processor with the fastest DNN block execution refers to its primary processor, while the slower refers to its secondary processor. We aim to optimize the

block execution on its secondary processor. However, optimizing for each block may cause a vast computation overhead for architecture search and also lead to high storage overhead for storing each optimized block. Hence, we choose to optimize the blocks that have high computational difference (e.g. $CD > \varepsilon$) as well as high workload proportion (e.g. $WP > 1/block_num$) to reduce those overheads. The thresholds for the high computational difference could be adjusted according to the user requirements. A lower computational difference threshold means that more blocks will be optimized, which will also result in heavy NAS workloads. In our experiments, we set it to 1 or 10 according to model types. Based on the profiling data, we calculate the computational difference (CD) of each block and the workload proportion (WP) of each block based on Eq. 1.

$$CD = \frac{T^{sec}(B_k)}{T^{pri}(B_k)}, \quad WP = \frac{T^{pri}(B_k)}{\sum_{k=0}^{k_{max}} T^{pri}(B_k)} \tag{1}$$

B_k refers to the kth block, and T^{pri}/ T^{sec} is the block execution time on the primary/secondary processor. This block selection mechanism avoids possible massive data transmission between CPU and GPU. If the feature maps of one DNN model are huge, it is not suitable for cross-processor inference in principle, and BlastNet will not partition it into blocks. However, BlastNet can handle DL task sets with different characteristics, and models that have large feature maps can still benefit from the optimization of other models with duo-blocks or the scheduling mechanism.

5.1.2 NAS-based Block Optimization. To optimize the inefficient DNN block for different processors, we first customize a pool of candidate blocks for each block to be optimized. We only consider convolutional and fully-connected layers since they account for the most execution time of the model inference according to previous works [12, 38]. For candidate blocks of a convolutional block, we choose the most CPU-friendly operators for optimization on CPU, i.e., depthwise separable convolutional layer [18]. As shown in Fig. 8, to maintain the same input size and output size as the original block, each candidate block has a head module and a tail module. The head/tail module is a lightweight convolutional layer with 1×1 kernel size, and has the same channel size as the prior/next block. Different candidate blocks have different layer numbers and channel sizes. Typically, we choose up to 10 depthwise separable convolutional layers and vary the channel size from 32 to 512 with exponential growth for a large search space. For the convolutional layer to be optimized to GPU, we adopt denser convolutional layers, which are more friendly for inference on GPU. For candidate blocks of fully-connected blocks, we choose the fully-connected layer with a smaller/larger channel size and stack them for optimization.

We then search for the optimal block that minimizes the accuracy loss while having less inference time on the secondary processor, which is formulated in Eq.2. The optimized block B_k^{new} can be a CPU block B_k^C if its secondary processor is CPU or a GPU block B_k^G if its secondary processor is GPU.

$$\begin{aligned} \forall\, k \quad & \min\ \mathcal{L}(B_k^{new}, W^*) \\ s.t.\ & T^{sec}(B_k^{old}) > T^{sec}(B_k^{new}) \end{aligned} \tag{2}$$

$\mathcal{L}(\cdot)$ is the loss function of the DNN model with k^{th} optimized block and the original model weight W^*. To solve the problem formulated in Eq.2, we first profile the inference time of each candidate block on the target processor and find the blocks that satisfy the

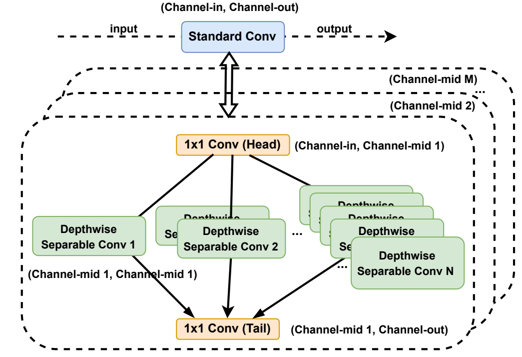


Figure 8: Example for candidate blocks of a convolutional layer on CPU

latency constraint, and then search for the block that minimizes the accuracy loss. The search algorithm is based on the Differentiable Architecture Search (DARTS) algorithm [34], which is one of the state-of-the-art one-shot NAS algorithms. The DARTS-based technique allows efficient architecture search by the gradient descent, which is orders of magnitude faster than state-of-the-art non-differentiable techniques. In order to ensure the compatibility of the optimized block with the original model, we fix the model weights of other blocks when calculating the architecture gradient and weight gradient in the DARTS-based search procedure and only retrain the weight of the searched optimal block.

Note that we optimize each individual block instead of optimizing all blocks at one time. Because searching all optimal blocks at one time can only guarantee the accuracy of the model when all optimized blocks are executed. However, as we will introduce in the next section, the DNN model may execute its kth block either B_k^G on the GPU or B_k^C on the CPU. Hence, the total execution sequence for an instance of DNN inference could be $B_0^C \rightarrow B_1^G, ..., \rightarrow B_k^C$, which not always contains all the optimized blocks.

5.2 Dynamic Cross-Processor Scheduling

Current DL frameworks such as PyTorch and TensorFlow can only allocate models to a single processor before execution. However, reallocating the whole model incurs significant overhead during model loading and initialization. We design a dynamic real-time cross-processor DNN model scheduling mechanism for concurrent DNNs execution. Fig. 9 illustrates the design. First, the scheduler prioritizes each duo-block based on its task urgency. Then, a primary-processor-first execution mechanism decides the execution processor for each duo-block based on the status of the processors. Our primary-processor-first execution mechanism also considers the accuracy loss caused by alternative execution of CPU/GPU blocks. Finally, we use a multi-worker thread executor with a duo-block memory manager to reduce the communication and memory copy overhead.

Task Model for DNN Inference. Each DNN model inference here consists of a sequence of block execution. Therefore, each DNN inference task τ_i ($i \in \{1, 2, ..., n\}$) comprises k sequential sub-tasks

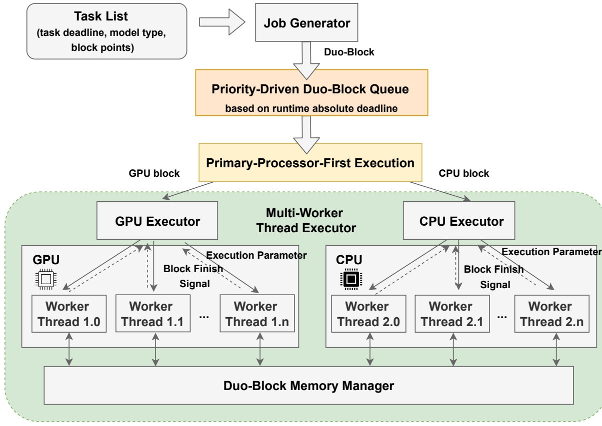


Figure 9: Procedure for cross-processor scheduling, worker thread 1.1 represents the worker thread for DNN model 1 on the GPU processor.

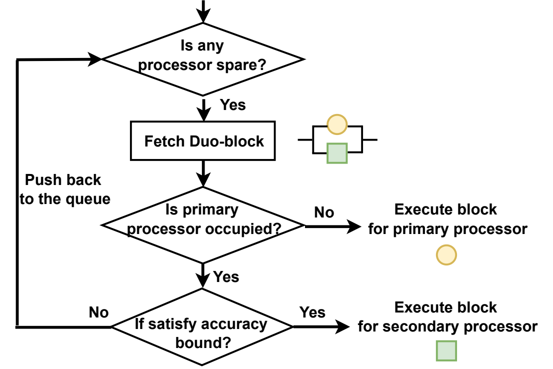


Figure 10: Primary-processor-first execution mechanism

$B_0 \rightarrow B_1, ..., \rightarrow B_k$, where k denotes the total number of blocks for DNN inference task τ_i. Since each duo-block has two structures, the kth block can be executed using either B_k^G on the GPU or B_k^C on the CPU. Real-world deep learning applications usually process input data periodically and require the results in time. For example, the DL task for traffic light detection is initiated at the same frequency as the camera's sampling (e.g., 10 fps). Therefore, each DNN inference task τ_i is a task with the period of T_i and a relative deadline D_i, where D_i can equal to T_i. Each task τ_i can be defined as an array, i.e., $(\{B_k^{G/C}\}, D_i, T_i)$.

DNN Duo-Block Scheduling. We adopt a *priority-driven duo-block queue* to prioritize the execution order of each duo-block. Specifically, when a DNN inference task boots, our job generator periodically generates the first sub-job (i.e., DNN duo-block B_0) for the task according to its period. We prioritize each duo-block according to the expected absolute completion time (i.e., the deadline) of its model inference: the earlier deadline, the higher priority. This design can help blocks from the most urgent tasks to be executed in time. The duo-block queue here is implemented as a priority queue to support priority-driven scheduling. The job generator pushes a new job to the duo-block queue only when the prior job is finished, or the job is discarded. This mechanism can avoid persistent job loss.

We design a *primary-processor-first execution* mechanism for duo-block execution to execute as many blocks on the primary processor as possible and manage the accuracy loss introduced by the optimized block. Fig. 10 shows the flow chart of the execution mechanism. We fetch the duo-block from the head of the queue once there are spare processors. If its primary processor is not occupied by other blocks, we execute the block for the primary processor in the duo-block. Note that a block for the primary processor is either a CPU block or a GPU block. Otherwise, we estimate the highest possible accuracy of executing the block for its secondary processor based on the prestored accuracy information for each execution path. We only execute this block when the accuracy meets a user-defined threshold, i.e., the minimum acceptable accuracy. Otherwise, it will be pushed back to the queue.

We also design a *multi-worker thread executor* to implement our DNN block scheduling approach. The processor-specific executor (i.e., GPU/CPU executor) dispatches the block to its corresponding DNN model worker thread for execution. Each worker thread has a fixed affinity for the CPU or GPU. If the worker thread needs to execute the first block of the DNN model (i.e., B_0), it will perform preprocessing for each DNN inference task, such as data fetching and image resizing for image classification tasks. Post-processing such as rendering and encoding will be performed after the execution of the last block. When initializing a DNN model worker thread, the worker thread preloads the DNN model architecture and weights into CPU/GPU memories according to its processor affinity. Without this mechanism, the execution needs to reload and initialize the DNN block every time and incurs extensive initialization and memory copy overheads. Our multi-worker thread executor also includes a *duo-block memory manager* to control the data transmission among multiple threads. Duo-block memory manager copies the data between blocks that execute on the different processors and only passes the pointers between blocks that execute on the same processors, which minimizes the data transmission between executed blocks.

6 EVALUATION

6.1 Implementation and Experiment Setup

We use a desktop-class (Intel i9 CPU + NVIDIA RTX 2080 GPU), and two edge platforms (NVIDIA AGX Xavier and NVIDIA Jetson TX2) with both CPU and GPU processors for evaluations. The hardware configurations are shown in Table 1. The details of the DNN inference tasks used in our experiments are shown in Table 2. We use PyTorch as the DL framework for model generation, NNI [45] as the NAS framework and LibTorch (C++ frontend of PyTorch) with CUDA library for DNN inference. We note that BlastNet can be ported to other DL frameworks like TensorFlow, MXNet, and MindSpore by converting the models to ONNX format for the C++ frontend.

Table 1: Platforms used in evaluation experiments.

Platform	GPU	CPU	Memory	Storage
NVIDIA AGX Xavier	512-core Volta	8-core ARMv8.2	16GB	32GB
NVIDIA Jetson TX2	256-core Pascal	2-core ARM Denver + 4-core ARM A57	8GB	32GB
Desktop	RTX2080	8-core Intel i9-9900K	32GB	5TB

Table 2: DNN inference tasks used in evaluation experiments.

DNN Inference Task Type	Dataset	DNN Model
Image Classification	CIFAR10 [24]	MobileNet[17], VGG11, AlexNet
Sign Recognition	GTSRB [48]	ResNet18
Object Detection	COCO [32]	YOLO [44]

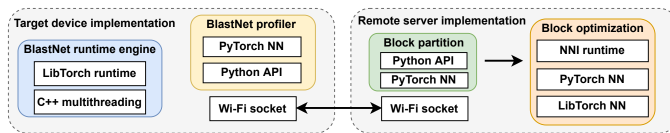


Figure 11: BlastNet Software Implementation

Figure 11 shows the system implementation of BlastNet, which includes a remote server for block optimization and the target device for on-device execution, which communicates via a standard socket. Specifically, the server first transmits the model files of candidate blocks to the client. After receiving the model files, the client profiles the candidate block on the target processor and then returns the profiling data to the server. In this way, while the profiling is performed on the target edge platform, more powerful servers can be utilized to efficiently conduct DNN block optimization. We convert the user-defined PyTorch model to the NNI model with candidate blocks, using an NNI-Pytorch model converter implemented by ourselves. It is also used to convert the model back to PyTorch when the NAS-based optimization is finished. The models are then cast to a LibTorch format to work with C++ code for DNN inference on the edge platforms.

In the implementation of the dynamic cross-processor scheduler, the inference of each model and the scheduler itself are executed in separate individual threads. The CPU affinity is fixed for each thread, and the edge platform is set to its maximum power mode. To support the block-level inference of each DNN model, we first modify the *forward* function of the model under the PyTorch framework, which supports flexible start and end points of model inference. Specifically, the start and end points of each block are stored offline, which can then be invoked by the scheduler to execute any given set of blocks.

6.2 End-to-end System Evaluation

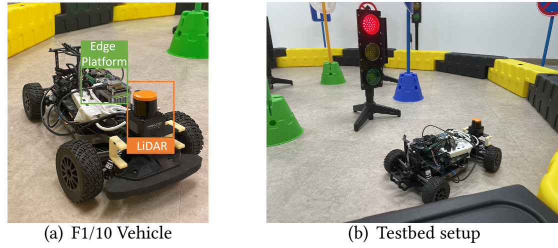
We implement BlastNet and evaluate the end-to-end system performance using an autonomous driving testbed. As shown in Fig. 12 (a), we use an F1/10 autonomous vehicle [5] equipping a LiDAR and an edge computing platform, i.e., NVIDIA TX2 with Orbitty Carrier board. The vehicle can detect the side of the road and follow along the lane fence with a dimension of $5m \times 6m$ while recognizing the traffic signs at the same time. There are four real-time DL tasks for traffic sign recognition and a lane detection task running on this platform. To simulate realistic outdoor traffic conditions, we preload the GTSRB dataset [48] on the SD card of the vehicle. It contains data of 43 signs captured by visual sensors. Four different ResNet18 models are run on the vehicle to recognize traffic signs. This is consistent with the common practice of autonomous driving systems, where multiple models are used to process the data of sensors installed at different positions of the vehicle [23]. We choose ResNet18 for evaluation because it is widely used as the backbone for most popular neural networks, such as RetinaNet [31] and DetNet [30].

We test various scenarios by running the F1/10 autonomous vehicle at different speeds, resulting in different levels of resource utilization. Specifically, the vehicle may remain static, runs at $1m/s$, or at $2m/s$. We run the experiment for 300 seconds for each setting which contains about 1500 jobs. We use the tegrastats toolkit [41] to measure the CPU utilization without DNN workload with a sampling rate of 1000 fps. Fig. 13(a) shows the average CPU utilization for a sliding time window of 100ms. It can be seen that the CPU utilization exhibits significant fluctuation as sensor data is being processed. Moreover, when the vehicle moves, the motion control induces a substantial CPU load.

We evaluate the performance of BlastNet by comparing it against the baseline *Layer_Sched*. *Layer_Sched* schedules the DNN model at the layer level, in which each sub-job refers to a single DNN layer. We use the deadline missing rate (i.e., Eq. 3) to quantify how well the real-time requirement is met. Deadline missing rate is a widely-used metric for real-time applications [1, 33]. With the joint consideration of car speed and the field of view of car cameras, we set the deadline for each task as 400ms, which is enough to detect an object in time. Fig. 13(b) shows the deadline missing rate of the task that yields the best performance for the baseline *Layer_Sched*) over three speeds. The results show that BlastNet consistently maintains a deadline missing rate below 5% for all the speed levels (in which the highest missing rate 4.25% was given by the high-speed $2m/s$). In contrast, *Layer_Sched* causes massive deadline misses (18.96% for static, 19.97% for high-speed).

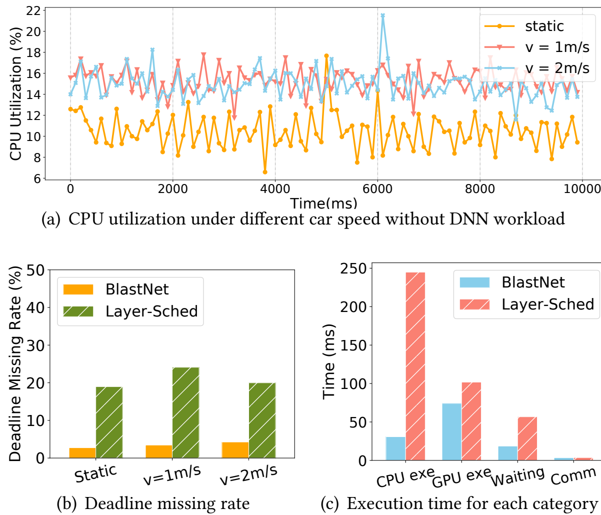
We further analyze the advantages of BlastNet by calculating the average CPU/GPU execution time, waiting time, and communication time per inference job. The waiting time here refers to the idle time from job release to completion, which may be caused by resource contention of other models. Fig. 13(c) shows the average latency for each delay category within a duration of $100s$. We observe that BlastNet effectively reduces the GPU execution time (74.08ms for BlastNet and 101.78 ms for *Layer_Sched*). This is because BlastNet executes more DNN workload efficiently on CPU (30.74ms for BlastNet and 244.84 ms for *Layer_Sched*). In addition, BlastNet also reduces the waiting time due to the effectiveness of cross-processor scheduling.

In conclusion, the evaluation in this section shows that BlastNet maintains consistent low deadline missing rates and is robust



(a) F1/10 Vehicle (b) Testbed setup

Figure 12: F1/10 autonomous driving testbed.



(a) CPU utilization under different car speed without DNN workload

(b) Deadline missing rate (c) Execution time for each category

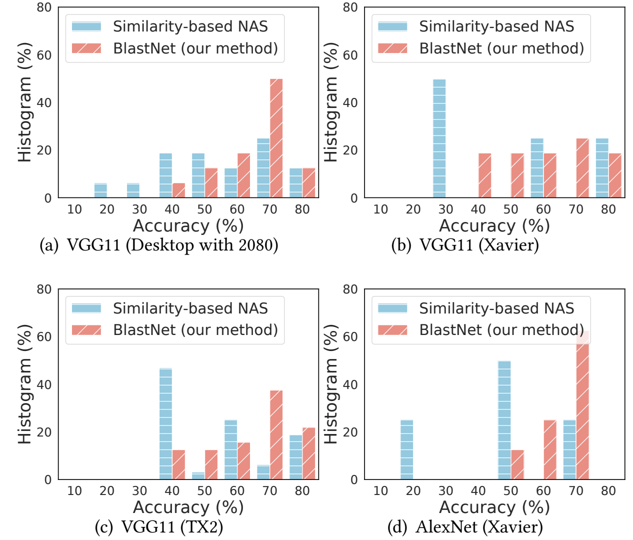
Figure 13: Performance of BlastNet under various driving settings.

to diverse conditions including different resource availability and driving speeds.

6.3 Performance of Cross-processor Duo-block Generation

We now evaluate the performance of our block-level DNN optimization on different platforms. To evaluate the effectiveness of the proposed block-level DNN optimization approach in BlastNet, we deploy a *Similarity-based NAS* as the baseline, which searches the optimized architecture based on the output difference between the optimized block and the original block. This approach is also adopted by the *DFN* [27] for searching efficient functions to replace the entire DNN model and *LegoDNN* [12] for retraining scaled DNN layers. We implement this baseline by modifying the loss function used in the neural architecture search. We use the Euclidean Distance to describe the similarity between the original block and the optimized block. We also use the same loss function for retraining the weights of optimized blocks.

We evaluate the accuracy of all the possible execution paths instead of the model with all the optimized blocks. This is because the DNN model may execute its block either on its primary processor or its secondary processor, which leads to a set of inference paths. Each inference path corresponds to an accuracy. Fig. 14 shows the distribution of the accuracy across all inference paths. There are nine accuracy bins in the x-axis, and each bin stands for a range of 10%, e.g. 30% means the accuracy range from 30% to 40%. We calculate the percentage of inference paths falling into each accuracy bin. Results show that BlastNet performs better than the *Similarity-based NAS* approach in terms of the accuracy distribution for all the inference paths. As shown in Fig. 14(a), for adapting VGG11 on a typical desktop-class platform, 81.25% of the inference paths are with the accuracies of more than 70%. While *Similarity-based NAS* only achieves 50% inference paths with the accuracies of more than 70%. Most of the inference paths of BlastNet are distributed in the high precision range, which guarantees high precision for most inferences. Fig. 14 also shows the accuracy distribution under other settings of different models and platforms. Compared with *Similarity-based NAS*, our approach can achieve less accuracy loss for all the paths of the optimized blocks and thus enable more inference with high accuracy.



(a) VGG11 (Desktop with 2080) (b) VGG11 (Xavier)

(c) VGG11 (TX2) (d) AlexNet (Xavier)

Figure 14: Model accuracy with all possible inference paths with the optimized blocks.

6.4 Overall Performance of BlastNet

To reflect the performance gain of block-level DNN optimization and scheduling, we compare our approach with four methods, including *Mono_Sched*, *Layer_Sched*, *Mono_Adapt*, and *BlastNet-w/o-PF*. Both baseline *Mono_Sched* and baseline *Layer_Sched* adopt the same priority-based queue (i.e., Earliest Deadline First) as BlastNet to prioritize the issued DNN inference task. Baseline *Mono_Sched* schedules the DNN models by monolithically allocating them to heterogeneous resources. Specifically, it dispatches the entire DNN model to a processor once there exists free resource on that processor. Baseline *Layer_Sched* schedules the DNN model at the layer

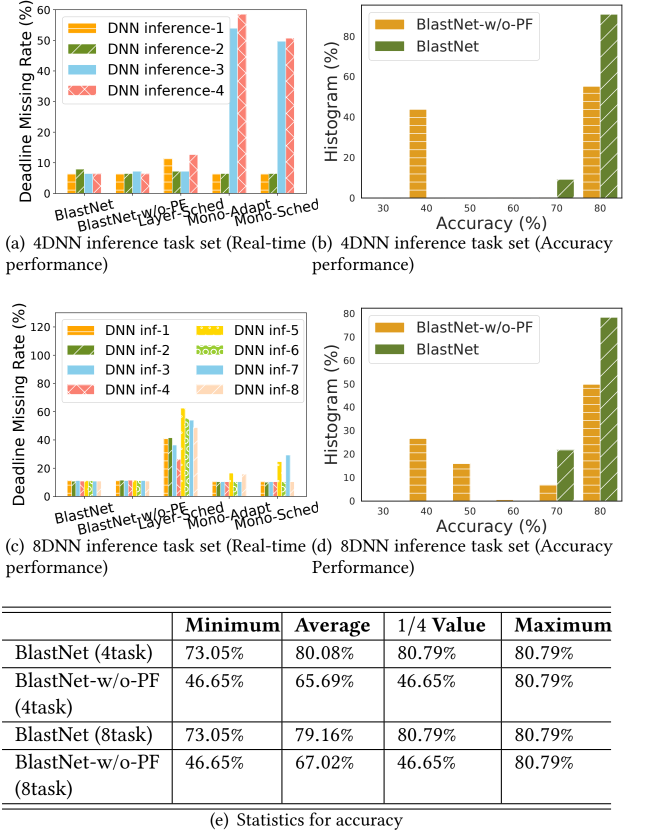
level, in which each sub-job refers to a single DNN layer. This baseline is similar to several existing works (e.g., *DART* [55]) that schedule DNN models at the layer level, as we introduced in Section 2. To evaluate the performance of our block-level DNN scheduling, we design a baseline *Mono_Adapt* that employs the whole model with all the optimized blocks for inference. Baseline *Mono_Adapt* differs from BlastNet only in DNN scheduling, i.e., it adopts monolithic scheduling (same as *Mono_Sched*) to execute the models. To evaluate the effectiveness of our primary-first execution mechanism, we design a baseline *BlastNet-w/o-PF*, which differs from BlastNet only in that it has no primary-first execution mechanism. We set 70% as the accuracy bound in our primary-first execution mechanism.

We use *deadline missing rate* to quantify how well the task real-time requirement (i.e., task deadline) is met under our method and other four baselines. The deadline missing rate as defined in Eq. 3 denotes the percentage of missed jobs among all inference jobs of a DNN inference task,

$$missing_rate = \frac{N^{overtime_job} + N^{skipped_job}}{N^{total_released_job}}, \tag{3}$$

where $N^{overtime_job}$ denotes the number of jobs that exceed their absolute deadlines, $N^{skipped_job}$ denotes the number of the skipped job due to the deadline missing of the previous job, and $N^{total_released_job}$ denotes the number of total released jobs. Each approach is evaluated for 120000 ms under the same configuration.

To show the accuracy performance of BlastNet and the other baselines, we adopt the same histogram setting in Section 6.3 as the metric. We show the probability for each accuracy range, which is calculated based on the accuracy of all DNN inferences, i.e., all the DNN inference instances for one task set. To quantify the accuracy performance, we also calculate the average accuracy loss by comparing the average inference accuracy and the accuracy of the original DNN model.



(a) 4DNN inference task set (Real-time performance) (b) 4DNN inference task set (Accuracy performance)

(c) 8DNN inference task set (Real-time performance) (d) 8DNN inference task set (Accuracy Performance)

	Minimum	Average	1/4 Value	Maximum
BlastNet (4task)	73.05%	80.08%	80.79%	80.79%
BlastNet-w/o-PF (4task)	46.65%	65.69%	46.65%	80.79%
BlastNet (8task)	73.05%	79.16%	80.79%	80.79%
BlastNet-w/o-PF (8task)	46.65%	67.02%	46.65%	80.79%

(e) Statistics for accuracy

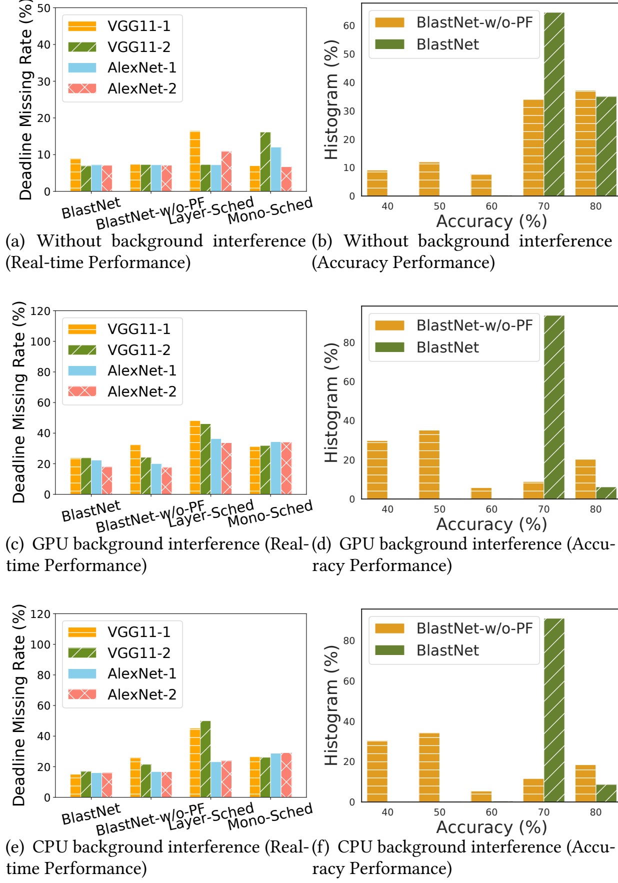
Figure 15: Real-time/Accuracy performance of BlastNet under different DNN workloads.

6.4.1 Impact of different DNN workloads. We demonstrate the effectiveness of BlastNet under different workloads induced by various settings of task numbers. Specifically, we consider low and high workloads for a task set with four and eight typical DNN inference tasks respectively. For unifying the workload of each DNN inference task, we adopt the same DNN model (i.e., VGG11) for each task. The deadline for each task is set to be four times the measured average inference time of a single DNN inference task. In order to show different real-time requirements, we randomly increase or decrease the deadline for each task by 2%.

As shown in Fig. 15(a), with light workload, BlastNet reduces the deadline missing rate of task VGG11-4 by 44.37% compared to *Mono_Sched*, 6.34% compared to *Layer_Sched*, and 52.11% compared to *Mono_Adapt* with no accuracy loss. As shown in Fig. 15(b), BlastNet achieves more inferences with high accuracy than *BlastNet-w/o-PF*. Specifically, compared with *BlastNet-w/o-PF*, BlastNet can sacrifice 14.39% less accuracy loss at average, while achieving comparable performance of deadline missing rate. For heavy workload as shown in Fig. 15(c), BlastNet still can achieve 10.63% average deadline missing rate with minor accuracy loss (*maximum* − *average* = 1.63%), while *Layer_Sched* can only achieve 45.70%. Although *Mono_Adapt* has a similar missing rate to ours (11.78%), our method avoids 32.51% average accuracy loss compared with *Mono_Adapt*. BlastNet achieves 12.14% reduction in accuracy loss compared with *BlastNet-w/o-PF* with comparable deadline missing rate (10.63%). BlastNet performs better than most other methods under both light and heavy workloads.

6.4.2 Impact of background load. We evaluate BlastNet under various GPU and CPU background loads to show the robustness of our system. We execute a YOLO model on the GPU or a MobileNet model on the CPU in a standalone thread as the GPU or CPU background load, respectively. We set the affinity of the working thread for executing our task set and the thread for background load to the same CPU cores. We use four DNN inference tasks with two types of typical DNN models. As a comparison, we also show the deadline missing rate and accuracy distribution of those four DNN inference tasks without background load in Fig. 16(a) and Fig. 16(b).

As shown in Fig. 16(c), under interference of GPU background workload, BlastNet can reduce the deadline missing rate by 14.62% and 11.31% compared with *Layer_Sched* and *Mono_Sched* with minor accuracy loss (7.31%). For background load interference on CPU, Fig. 16(e) shows that BlastNet still can achieve 15.99% average deadline missing rate with 7.31% average accuracy loss, while *Layer_Sched* and *Mono_Sched* only achieve 35.54% and 27.65% respectively. Comparing the results in Fig. 16(b) and Fig. 16(d),16(f), we also observe a drop of probability for high accuracy. This is

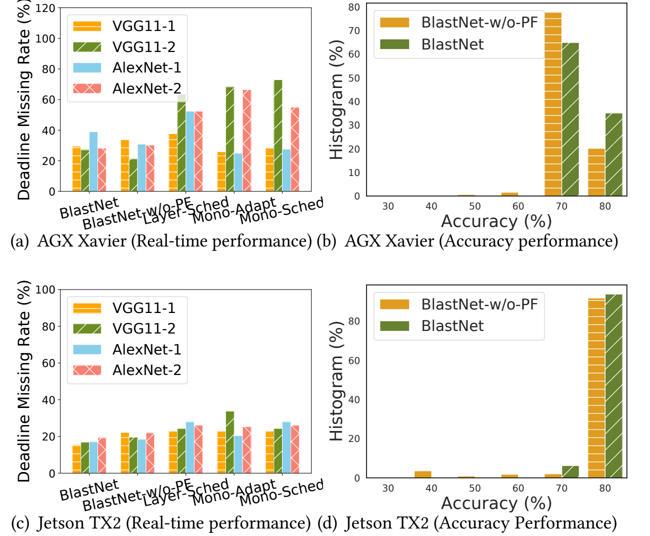


(a) Without background interference (Real-time Performance)
(b) Without background interference (Accuracy Performance)
(c) GPU background interference (Real-time Performance)
(d) GPU background interference (Accuracy Performance)
(e) CPU background interference (Real-time Performance)
(f) CPU background interference (Accuracy Performance)

Figure 16: Real-time/Accuracy performance of BlastNet under interference.

because our scheduler tends to offload DNN workloads to the secondary processor when the primary processor has a heavy workload to guarantee the real-time performance of DNN model inference. This result shows that BlastNet achieves better performance over baselines under both GPU background load interference and CPU background load interference.

6.4.3 Different edge platforms. In this section, we assess the generality of BlastNet for various edge platforms. The experiments in previous sections are conducted on a desktop-class platform. In this section, we evaluate BlastNet on low-power edge devices. As shown in Fig. 17(a),17(b) and Fig. 17(c),17(d), BlastNet can reduce the deadline missing rate effectively on both edge platforms with minor accuracy loss. Specifically, BlastNet suffers a minor accuracy loss 1.54% on Xavier, while reducing deadline missing rate by 20.57%, 15.45%, 15.04% on average, compared with *Layer_Sched*, *Mono_Adapt* and *Mono_Sched* algorithms respectively. On TX2, BlastNet achieves 1.63% reduction in accuracy loss compared with the baseline (*BlastNet-w/o-PF*) while achieving comparable average deadline missing rate (17.13% for BlastNet, 20.51% for *BlastNet-w/o-PF*). This result shows that BlastNet can be effectively applied on different edge platforms.



(a) AGX Xavier (Real-time performance)
(b) AGX Xavier (Accuracy performance)
(c) Jetson TX2 (Real-time performance)
(d) Jetson TX2 (Accuracy Performance)

Figure 17: Performance comparison of BlastNet under different edge platforms.

6.4.4 CPU/GPU utilization. We trace the execution of VGG11 using BlastNet and the baseline to evaluate the CPU/GPU utilization and communication overhead on an NVIDIA Jetson TX2. We classify the operations into three categories, i.e., inference, communication, and idle. Fig. 18 shows the timelines of the operations in a time duration of 2s and the distribution of time. Communication time describes the time of copying input data between different processors for each block/layer. Note that this copy only occurs when the current and subsequent blocks/layers are executed on different processors. We also calculate the proportion of the total time for each part. We observe that BlastNet uses 90.95% of GPU time for inference while only 70.20% is used by *Layer_Sched*. We observe that the communication overhead caused by *Layer_Sched* is almost the same as BlastNet. This is because BlastNet executes more jobs within the same time and leads to more frequent data transmission. In addition, we can find from the timeline that *Layer_Sched* incurs a longer communication time for single data transmission. The results show that BlastNet can effectively and efficiently reduce the block time caused by unbalanced inference performance between different processors, thus improving overall CPU/GPU utilization and lowering the deadline missing rate.

6.5 System Overhead

Table 3: CPU Overhead of Block-level DNN Scheduling

Task Set Size	Desktop	Xavier	TX2
2 DNN inference tasks	1.21%	1.61%	2.86%
4 DNN inference tasks	1.52%	1.38%	3.97%
6 DNN inference tasks	2.08 %	1.71%	3.20%

The major system overhead of BlastNet is caused by the dynamic cross-processor DNN scheduler since it needs to run online. The

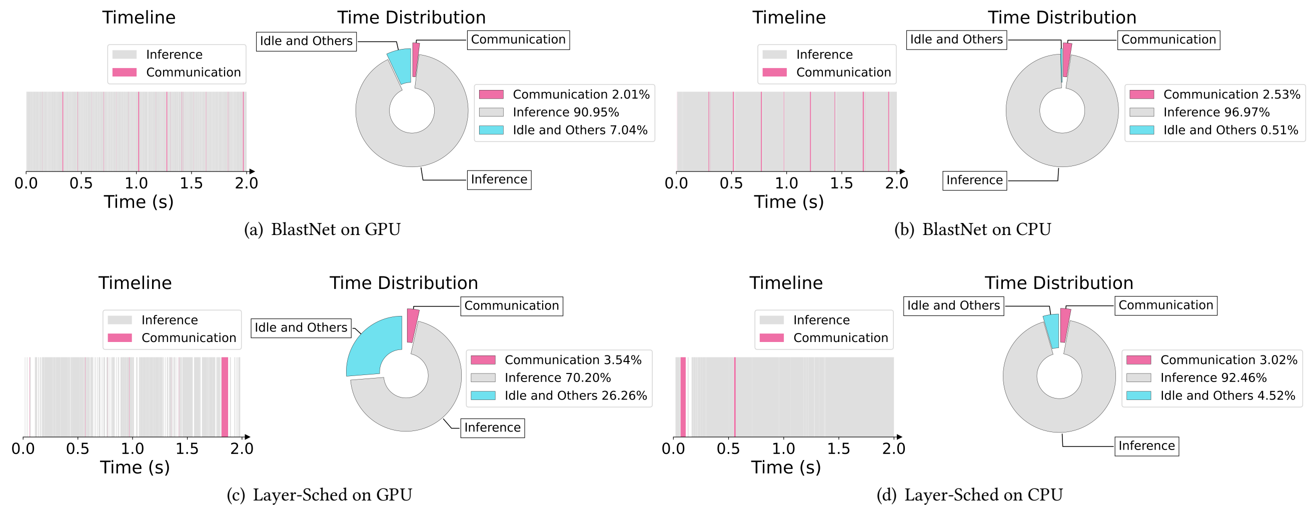


Figure 18: Execution timeline for inference, communication, and idle.

cross-processor block generation may also incur considerable overhead. However, it is offline and can also be offloaded to the cloud. Therefore, we focus on analyzing the CPU overhead of our block-level DNN scheduler in this section. We measure the overhead by setting the CPU affinity of our scheduler/optimizer to a single CPU core. Table 3 shows the incurred CPU overhead (measured as the percentage of CPU usage) by our scheduler. The majority of scheduling overhead is due to the control of DNN block execution. As shown in Table 3, when the number of DNN inference tasks increases, CPU overhead remains almost the same. Especially, due to the different processing capabilities of CPU cores as shown in Section 6.1, CPU overhead is different among platforms.

7 DISCUSSION

Scalability to New Platforms. BlastNet can be extended to support a wider range of heterogeneous architectures such as CPU-NPU [16] and GPU-FPGA [14]. While our basic idea of optimizing processor-specific blocks is still valid, the NAS technique adopted in BlastNet may lead to a vast migration overhead for a new device. In such a case, BlastNet may integrate few-shot transfer learning methods [52] with the cross-processor block generation to accelerate the adaptation procedure to a new target hardware platform. Specifically, we will design a latency predictor to predict the execution time of a DNN architecture, which speeds up the convergence of neural architecture search. This predictor can quickly adapt to a new platform based on the few-shot learning technique.

Uncertain Workload. We evaluate BlastNet under periodic DNN model inference to simulate real-time applications that require sensors to be sampled at a fixed frequency. Besides periodic DNN model inference, BlastNet can also accommodate uncertain dynamic workloads, e.g., tasks arriving over time such as voice control triggered by keyword spotting [3]. The dynamic model scheduling mechanism adopted in BlastNet can handle such uncertain workloads via the priority-driven job queues. For instance, the priorities of opportunistic tasks may be assigned based on their absolute deadlines.

Application Scenarios. Compared with model-level scheduling, the block-level scheduling approach will enable BlastNet to perform better for the mixed tasksets with big and small models. This is because, under model-level execution, the execution of a big model may block the execution of a small model, which can be relieved by the more fine-grained block-level execution. Moreover, BlastNet is also effective for the DL tasks that perform well on a single processor (e.g., all GPU-friendly models), since the NAS-based block optimization in BlastNet can adapt the original model to a new processor efficiently in a fine granularity.

8 CONCLUSION

In this paper, we present a new system named BlastNet, which supports concurrent real-time DNN model inference on CPU-GPU heterogeneous edge platforms through block-level model optimization and scheduling. Based on duo-block - a new model optimization and scheduling abstraction, BlastNet integrates novel techniques at both design time and runtime to optimize the architecture of DNN models in block-level granularity and dynamically schedule concurrent block inference on CPU-GPU heterogeneous resources. We implement BlastNet on three mainstream heterogeneous CPU-GPU edge platforms and an indoor autonomous driving testbed. Our extensive experiments show that BlastNet can enable concurrent real-time DNN model inference to achieve satisfactory real-time performance. Moreover, compared to several state-of-the-art baselines with fine-/coarse-grained scheduling policies, BlastNet can reduce deadline missing rate up to 35.07% while only sacrificing negligible DNN model accuracy. Therefore, BlastNet can support a wide range of emerging data-intensive and time-critical applications running on edge platforms.

ACKNOWLEDGEMENT

The work described in this article was partially supported by the Research Grants Council (RGC)-General Research Fund under Grant No. 14209619 and Grant No. 14203420.

REFERENCES

[1] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 107–118. IEEE, 2018.

[2] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[3] Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4087–4091. IEEE, 2014.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[5] F1TENTH Community. F1tenth. https://f1tenth.org/.

[6] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.

[7] Xianzhi Du, Mostafa El-Khamy, Jungwon Lee, and Larry Davis. Fused dnn: A deep neural network fusion approach to fast and robust pedestrian detection. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 953–961. IEEE, 2017.

[8] Alireza Ghaffari and Yvon Savaria. Cnn2gate: Toward designing a general framework for implementation of convolutional neural networks on fpga. *arXiv preprint arXiv:2004.04641*, 2020.

[9] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[10] News Google. Google tensor soc, titan m2 security chip features detailed. https://www.fonearena.com/blog/350642/google-tensor-soc-features.html/.

[11] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–177. IEEE, 2019.

[12] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y Chen. Legodnn: block-grained scaling of deep neural networks for mobile vision. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 406–419, 2021.

[13] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[14] Cong Hao, Atif Sarwari, Zhijie Jin, Husam Abu-Haimed, Daryl Sew, Yuhong Li, Xinheng Liu, Bryan Wu, Dongdong Fu, Junli Gu, et al. A hybrid gpu+ fpga system design for autonomous driving cars. In *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 121–126. IEEE, 2019.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 133–136. IEEE, 2020.

[17] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[18] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[19] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.

[20] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 209–221, 2022.

[21] Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 329–341. IEEE, 2021.

[22] Dewant Katare and Mohamed El-Sharkawy. Embedded system enabled vehicle collision detection: an ann classifier. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0284–0289. IEEE, 2019.

[23] Sandra Khvoynitskaya. 3 types of autonomous vehicle sensors in self-driving cars. https://www.itransition.com/blog/autonomous-vehicle-sensors.

[24] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[26] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.

[27] Seulki Lee and Shahriar Nirjon. Deep functional network (dfn) functional interpretation of deep neural networks for intelligent sensing systems. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*, pages 191–206, 2021.

[28] Peiliang Li, Xiaozhi Chen, and Shaojie Shen. Stereo r-cnn based 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7644–7652, 2019.

[29] Peilun Li, Guozhen Li, Zhangxi Yan, Youzeng Li, Meiqi Lu, Pengfei Xu, Yang Gu, Bing Bai, Yifei Zhang, and DiDi Chuxing. Spatio-temporal consistency and hierarchical matching for multi-target multi-camera vehicle tracking. In *CVPR Workshops*, pages 222–230, 2019.

[30] Zeming Li, Chao Peng, Gang Yu, Xiangyu Zhang, Yangdong Deng, and Jian Sun. Detnet: A backbone network for object detection. *arXiv preprint arXiv:1804.06215*, 2018.

[31] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[32] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[33] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2021.

[34] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[35] Xing Liu, Minjie Zhang, Chengming Zou, Jianfeng Yang, and Xin Yan. Edge intelligence for smart metro systems: Architecture and enabling technologies. *IEEE Network*, 36(1):136–143, 2021.

[36] S Divya Meena and Agilandeeswari Loganathan. Intelligent animal detection system using sparse multi discriminative-neural network (smd-nn) to mitigate animal-vehicle collision. *Environmental Science and Pollution Research*, 27(31):39619–39634, 2020.

[37] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[38] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[39] NVIDIA. Cuda c/c++ streams and concurrency. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[40] NVIDIA. Jetson agx xavier series modules and developer kit. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[41] NVIDIA. Tegrastats utility. https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html.

[42] PyTorch. Cpu threading and torchscript inference. https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html.

[43] Dipankar Raychaudhuri, Ivan Seskar, Gil Zussman, Thanasis Korakis, Dan Kilper, Tingjun Chen, Jakub Kolodziejski, Michael Sherman, Zoran Kostic, Xiaoxiong Gu, et al. Challenge: Cosmos: A city-scale programmable testbed for experimentation with advanced wireless. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–13, 2020.

[44] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[45] Microsoft Research. Nni (neural network intelligence). https://nni.readthedocs.io/en/stable/.

[46] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[47] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2017.

[48] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, (0):–, 2012.

[49] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[50] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. To bridge neural network design and real-world performance: A behaviour study for neural networks. *Proceedings of Machine Learning and Systems*, 3:21–37, 2021.

[51] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 215–228, 2021.

[52] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020.

[53] Wikipedia. Worst-case execution time. https://en.wikipedia.org/wiki/Worst-case_execution_time.

[54] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[55] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[56] Lele Xie, Tasweer Ahmad, Lianwen Jin, Yuliang Liu, and Sheng Zhang. A new cnn-based method for multi-directional car license plate detection. *IEEE Transactions on Intelligent Transportation Systems*, 19(2):507–517, 2018.

[57] Xiufeng Xie and Kyu-Han Kim. Source compression with bounded dnn perception loss for iot edge computer vision. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[58] Zhiyuan Xu, Dejun Yang, Chengxiang Yin, Jian Tang, Yanzhi Wang, and Guoliang Xue. A co-scheduling framework for dnn models on mobile and edge devices with heterogeneous hardware. *IEEE Transactions on Mobile Computing*, 2021.

[59] Juheon Yi and Youngki Lee. Heimdall: mobile gpu coordination platform for augmented reality applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[60] Fotios Zantalis, Grigorios Koulouras, Sotiris Karabetsos, and Dionisis Kandris. A review of machine learning and iot in smart transportation. *Future Internet*, 11(4):94, 2019.

[61] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, 2020.

[62] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. Ecrt: An edge computing system for real-time image-based object tracking. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 394–395, 2018.

[63] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. Edgeml: An automl framework for real-time deep learning on the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pages 133–144, 2021.