Arch2End: Two-stage Unified System-level Modeling for Heterogeneous Intelligent Devices

Weihong Liu, Zongwei Zhu*, Boyu Li, Yi Xiong, Zirui Lian, Jiawei Geng, and Xuehai Zhou, *Member, IEEE,*

***Abstract*—The surge in intelligent edge computing has propelled the adoption and expansion of Distributed Embedded Systems (DES). Numerous scheduling strategies are introduced to improve the DES throughput, such as latency-aware and group-based hierarchical scheduling. Effective device modeling can help in modular and plug-in scheduler design. For uniformity in scheduling interfaces, a unified device performance modeling is adopted, typically involving system-level modeling that incorporates both hardware and software stacks, broadly divided into two categories. Fine-grained modeling methods based on hardware architecture analysis become very difficult when dealing with a large number of heterogeneous devices, mainly because much architecture information is closed-source and costly to analyze. Coarse-grained methods are based on limited architecture information or benchmark models, resulting in insufficient generalization in the complex inference performance of diverse DNNs. Therefore, we introduce a two-stage system-level modeling method (Arch2End), combining limited architecture information with scalable benchmark models to achieve a unified performance representation. Stage one leverages public information to analyze architectures in a uniform abstraction and to design benchmark models for exploring device performance boundaries, ensuring uniformity. Stage two extracts critical device features from end-to-end inference metrics of extensive simulation models, ensuring universality and enhancing characterization capacity. Compared to the state-of-the-art methods, Arch2End achieves the lowest DNN latency prediction relative errors in the NAS-Bench-201 (1.7%) and real-world DNNs (8.2%). It also showcases superior performance in inter-group balanced device grouping strategies.**

***Index Terms*—distributed embedded system, scheduling, heterogeneous device modeling, hardware architecture, end-to-end inference, latency prediction, device grouping**

I. Introduction

Artificial Intelligent (AI) edge computing offloads many Deep Neural Network (DNN) tasks to embedded devices closer to data sources to alleviate the computation and communication burdens on the cloud [1], [2]. Distributed Embedded Systems (DES), composed of multiple embedded devices, enhance the performance of processing local data in AI edge scenarios through cooperative scheduling among devices [3]. As DES scales up, various performance-aware scheduling strategies are introduced to enhance the DES throughput [4], [5], outperforming the non-performance-aware strategies [6].

In recent years, the implementation of DES schedulers has tended to be modular and plug-in, necessitating unified and device-independent interfaces [7]. However, as the embedded device ecosystem becomes increasingly diverse [8], heterogeneous devices often employ specific analysis methods for performance awareness [9]–[12], posing significant challenges to designing unified interfaces. Thus, providing a unified interface through unified device modeling is critical to facilitate the understanding of device performance [13], [14]. Effective device modeling captures the unique characteristics of each device type, facilitating the specification of heterogeneous devices, the unified design of scheduler interfaces, and the rapid validation of scheduling strategies [15]. For instance, with a manageable number of tasks (m) and device types (n), unified device modeling aids schedulers in designing consistent interfaces to perceive heterogeneous device performance and predict DNN inference latencies, serving as a basis for scheduling [16]; with significantly large m and n, the costs of task scheduling and latency acquisition escalates. Hierarchical scheduling reduces complexity by dividing autonomous regions, such as device grouping in Pigeon [17]. unified device modeling provides standardized performance metrics for inter-group balanced device grouping strategies.

In recent years, device modeling methods have been broadly categorized into two types based on the depth of analysis of the hardware architecture: fine-grained and coarse-grained.

Fine-grained modeling methods characterize device performance by delving into hardware architecture features and operational implementation details, invariably entailing high analysis costs and dependence on detailed device information. Some work such as Timeloop [9], MAESTRO [10], and nn-Meter [11] focus on the detailed analysis of DNN operator implementations and inference processes, including data mapping, memory reuse, causality, and operator fusion optimization techniques. Such analyses require thorough hardware architecture information and entail high costs, with nn-Meter reporting an analysis time cost of up to several days per device [11]. More seriously, hardware design details of intelligent devices are often considered proprietary by manufacturers and are not publicly disclosed, thus hindering such analyses.

Coarse-grained modeling methods utilize publicly available architecture configurations or end-to-end measured met-

This work was supported by the National Key R&D Program of China (2022YFB4501600), the National Natural Science Foundation of China (No. 62102390), the special fund for Jiangsu Natural Resources Development (Innovation Project of Marine Science and Technology, No. JSZRHYKJ202218), and the National Key Laboratory of Science and Technology on Space Microwave (No.HTKJ2022KL504021). *(Corresponding author: Zongwei Zhu.)*

Weihong Liu, Yi Xiong, Zirui Lian, Jiawei Geng and Xuehai Zhou are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China and also with the Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China (e-mail: {lwh2017, xiongyi, ustclzr, gjw1998}@mail.ustc.edu.cn, xhzhou@ustc.edu.cn).

Zongwei Zhu and Boyu Li are with the School of Software Engineering, University of Science and Technology of China, Hefei 230026, China and also with the Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China (e-mail: zzw1988@ustc.edu.cn, llbbyy@mail.ustc.edu.cn).

rics to evaluate device performance fairly, circumventing the significant overhead associated with in-depth architecture analysis. However, such methods fail to accurately represent the complex inference performance of DNNs on heterogeneous devices. For instance, the instruction-level Roofline model [18] characterizes device differences by analyzing peak instruction throughput during computation, offering a unified yet imprecise perspective. PCN [16] and Glimpse [19] rely solely on available hardware configurations for predicting DNN latency but fail to represent devices with different architectures uniformly (e.g., GPUs and NPUs). Either expansion or merging of configuration vectors leads to a significant reduction in the device characterization capability. MLPerf [20], AIPerf [21], and HELP [22] have introduced benchmark suites for heterogeneous devices to evaluate performance comparisons of specific DNNs fairly, such as ResNet-50 and MobileNet-v1. However, their limited benchmark models cannot cover DNNs with different network structures or operators.

To simultaneously tackle the challenge of different architecture configurations and the limited coverage of benchmark models, we are motivated to combine both for a more comprehensive evaluation of heterogeneous device performance. Our primary objectives are to 1) design benchmark models based on limited architecture information for a unified coarse-grained representation of devices and 2) develop a generator to extend the coverage of benchmark models, extracting a unified representation of devices from extensive models.

Therefore, this paper presents a two-stage system-level modeling method (Arch2End) for heterogeneous devices to tackle significant architectural differences and partially disclosed device information. Stage one abstracts performance differences of heterogeneous devices into three dimensions and categorizes benchmark models based on partial information, roughly delineating the performance differences of devices under diverse boundary conditions. Stage two extracts black-box features from extensive simulated model inference metrics to compensate for the lack of performance perception due to unknown device information. Ultimately, Arch2End integrates features from both stages to support downstream tasks.

In summary, our main contributions are as follows:

- The proposed Arch2End is a unified device modeling method, integrating public architecture information and end-to-end inference metrics to deliver generalized and precise performance vectors for heterogeneous devices.
- Leveraging public device information, we analyze device performance from the architecture's computation, memory, and communication dimensions. The types of benchmark models for each dimension are defined and designed to systematically explore the performance boundaries.
- Utilizing a scalable simulation model generator, we apply the linear fitting technique to reduce the dimensionality of extensive simulation model metrics, thus providing a comprehensive representation of device performance via a standardized vector.
- Arch2End is evaluated on GPUs and NPUs. It achieves the lowest DNN latency prediction relative errors in the NAS-Bench-201 (1.7%) and the real-world DNNs (8.2%) and outperforms in inter-group balanced device grouping.

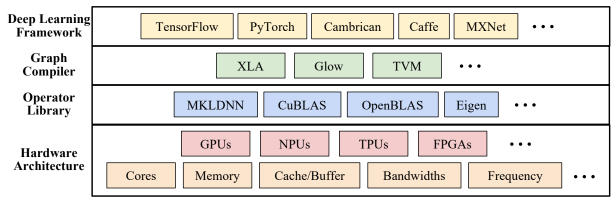


Fig. 1. Device system stack containing multiple heterogeneous dimensions. Hardware architecture includes microarchitecture and configuration.

II. Motivation

A. Limitations of Fine-grained Modeling Methods on Heterogeneous Devices

In pursuit of high-performance DNN inference in DES, over 100 companies are currently dedicated to advancing custom hardware architectures and software frameworks [23]. As illustrated in Fig. 1, dimensions such as hardware architecture, operator library, graph compiler, and deep learning framework offer a wide array of choices [24], [25]. The combinations of these options result in numerous heterogeneous devices with varying performance levels. The intricate heterogeneity and lack of publicly available information about these devices limit users' understanding of the devices they intend to model and evaluate, preventing in-depth feature analysis.

Most fine-grained modeling mothods are restricted by their reliance on invasive analysis tailored to specific heterogeneous dimensions in Table I. The application of these methods in heterogeneous scheduling scenarios faces several challenges:

Hardware Information Support: Invasive modeling is typically associated with hardware architecture information; however, both excessive reliance and insufficient consideration are inappropriate. For instance, Timeloop [9] and MAESTRO [10] require detailed architecture information, such as configuration information of components and data flow patterns, to model device performance, which is impractical for devices with undisclosed information. In contrast, nn-Meter [11] designs kernel-level test cases that do not depend on architecture information, potentially failing to explore the performance boundaries illustrated in Fig. 5 due to the omission of device-specific cases.

Operator Library Support: The increasing complexity of operator implementations poses challenges for device performance modeling. Techniques like operator fusion significantly enrich the operator library and enhance inference performance. The nn-Meter [11] develops specific performance prediction models for each device to account for variations in the operator library dimension. However, gathering substantial device performance data to train these models requires extensive analysis efforts and time costs (up to 4.4 days per device) [11]. Furthermore, such specific prediction models contradict the premise of unified device modeling.

These challenges lead to a reliance on detailed architectural information and significant costs, highlighting the impracticality of these methods in heterogeneous scenarios requiring unified and generalized device performance modeling. Furthermore, the commercial applications of intelligent devices often avoid delving into specific heterogeneous dimensions, opting instead to model devices comprehensively [15].

TABLE I
DESCRIPTION OF ARCHITECTURE INFORMATION AND INTRUSIVE ANALYSIS REQUIRED FOR FINE-GRAINED MODELING METHODS.

Methods	Architecture information	Intrusive analysis of operator realization processes
Timeloop [9]	Including the number of processing elements (PEs), the size of hierarchical memory resources, and the structure of interconnection networks.	**Data flow**: how the accelerator schedules operations and data under different architecture configurations?
MAESTRO [10]		**Data mapping**: how to efficiently map and execute DNNs on a given architecture configuration, involving data reuse and execution order?
nn-Meter [11]		**Operator library**: it focuses on analyzing graph optimization techniques for different devices, such as operator fusion.

Challenge 1. The device modeling process should be based on public information to describe system-level performance of devices, including hardware and its bundled software stack.

B. Inadequate Device Characterization for Coarse-grained Modeling Methods

NAS-Bench-201 [26], a public dataset of diverse neural networks, is frequently used to evaluate the representational capabilities of heterogeneous device modeling through multi-device latency prediction accuracy. We can contrast the representational capacity of different device modeling results by varying the device feature inputs to the prediction model.

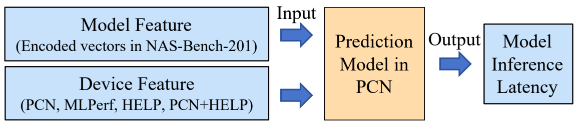


Fig. 2. Model latency prediction. The prediction model employs a neural network with a self-attention module, as mentioned in PCN [16], employing model features and device features as inputs and inference latencies as outputs.

The prediction experiment is shown in Fig. 2, and the relative prediction error is assessed via *Absolute Percentage Error* (APE) [27], i.e., $|\frac{A_t - P_t}{A_t}|$. A_t and P_t denote the actual and predicted values at data point t, respectively. The experiments compare the model latency prediction *APE* of different device modeling methods in heterogeneous devices containing GPUs and NPUs, whose details are elaborated in Section IV-A.

Due to the incomplete public disclosure of architecture information, many architecture-aware methods such as PCN [16] and Glimpse [19] rely solely on publicly available architecture configurations from technical manuals for device modeling. In the case of PCN, the *APE* of NAS-Bench-201 is shown in the upper part of Fig. 3. When heterogeneous devices are composed solely of GPUs or NPUs, the prediction accuracy within a 10% *APE* range can reach 97.1% and 96.3%, respectively. However, when heterogeneous devices include a combination of GPUs and NPUs, the prediction accuracy within the same error range drops to 85.9%. This decrease is attributed to the limited architecture configuration's inability to represent complex heterogeneous devices, affecting latency prediction accuracy across diverse DNNs.

Similarly, as shown in the lower part of Fig. 3, device modeling methods based on end-to-end measured metrics such as MLPerf [20] and HELP [22] also show lower accuracy, with only 72.3% and 86.8% accuracy within a 10% *APE* range, respectively. Although the real-world model inference latency directly reflects device performance, the limited coverage of test cases selected by such methods fails to characterize device performance across diverse DNNs. Surprisingly, we find that combining architecture information with end-to-end measured

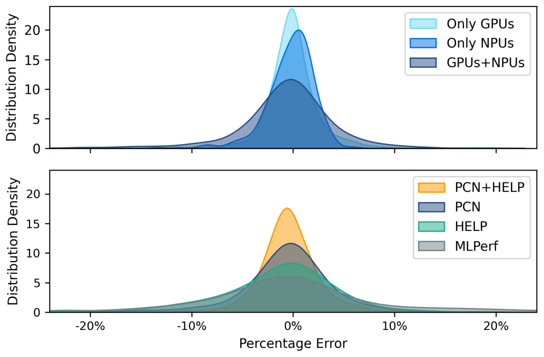


Fig. 3. Relative error distributions of latency prediction for NAS-Bench-201. The upper part represents the application of PCN on different sets of devices, and the lower part represents the application of different modeling methods on a set of GPUs and NPUs.

data yields better prediction results, achieving 93.1% accuracy within a 10% *APE* range. This suggests that architecture information and end-to-end inference metrics are complementary in characterizing device performance.

Challenge 2. Due to limitations in architecture information and the coverage of benchmark models, it is challenging to ensure the universality of coarse-grained modeling methods across diverse devices and DNNs, resulting in lower DNN latency prediction accuracy. Effectively integrating architecture information and end-to-end inference metrics for modeling presents a viable solution to this challenge.

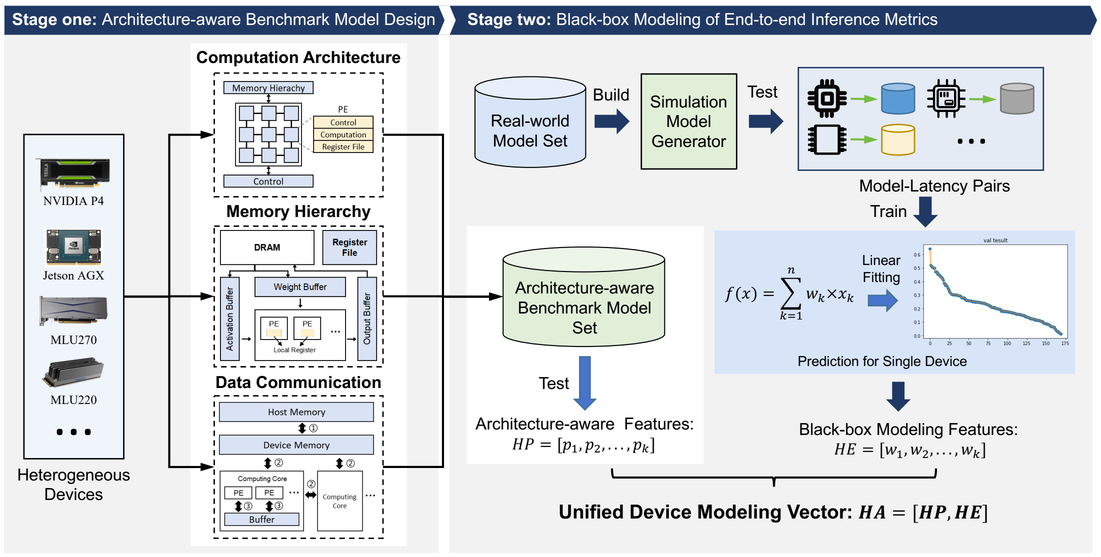
In summary, fine-grained modeling methods are significantly limited due to an overreliance on hardware architecture information. Meanwhile, although generalizable across heterogeneous devices, coarse-grained modeling methods face substantial challenges in performance modeling for diverse DNNs due to the limitations in architecture information and the coverage of benchmark models. Therefore, how to effectively integrate these two types of characteristics to develop a better and more universal device modeling method is the problem that Arch2End is proposed to address in this paper.

III. ARCH2END DESIGN

This section introduces a two-stage unified system-level modeling method (Arch2End) for heterogeneous devices and the details of its two stages.

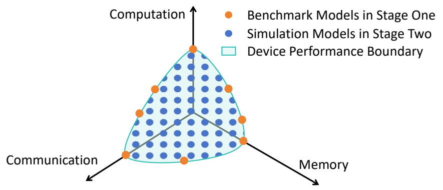
A. Overview Design

Fig. 4 illustrates the two-stage unified system-level modeling method (Arch2End) for heterogeneous devices, encompassing architecture-aware benchmark model design and black-box modeling of end-to-end inference metrics. Innovatively, Arch2End abstracts device performance into a three-dimensional space composed of computation, storage, and



Stage one: Architecture-aware Benchmark Model Design
Stage two: Black-box Modeling of End-to-end Inference Metrics
Computation Architecture
Memory Hierarchy
Data Communication
NVIDIA P4
Jetson AGX
MLU270
MLU220
Heterogeneous Devices
Real-world Model Set
Build
Simulation Model Generator
Test
Model-Latency Pairs
Train
Architecture-aware Benchmark Model Set
Test
Architecture-aware Features: $HP = [p_1, p_2, \ldots, p_k]$
$f(x) = \sum_{k=1}^{n} w_k \times x_k$
Linear Fitting
Prediction for Single Device
Black-box Modeling Features: $HE = [w_1, w_2, \ldots, w_k]$
Unified Device Modeling Vector: $HA = [HP, HE]$

Fig. 4. An overview of Arch2End. Stage one, positioned on the left, analyzes various architectures across three dimensions and designs corresponding benchmark models, using their inference latencies as architecture-aware features. Stage two, located on the right, introduces a scalable simulation model generator to gather extensive latency data and extracts device black-box modeling features based on linear fitting.



Computation
Communication
Memory
● Benchmark Models in Stage One
● Simulation Models in Stage Two
Device Performance Boundary

Fig. 5. The positions of benchmark and simulation models within the device performance boundary.

communication, as depicted in Fig. 5. The inference of DNNs is represented as probing the device's capabilities across these three dimensions (i.e., a point in the space). Thus, comprehensive model sampling within the device performance boundaries, supported by their inference metrics, serves as the basic data for device performance modeling.

As shown on the left side of Fig. 4, stage one, abstracts intelligent devices using limited available information to cope with significant architectural variations and undisclosed details among heterogeneous devices. It analyzes various architectures from computation, memory, and communication dimensions. **Benchmark models** corresponding to these three dimensions are designed to explore the performance boundaries of different devices, maximizing the distinction of device performance differences and taking the model's inference latency as an architecture-aware feature. As shown on the right side of Fig. 4, stage two introduces a scalable **simulation model** generator based on a real-world model set, enabling random sampling within the boundary to compensate for the insufficient sampling rate of benchmark models in space. To handle the extensive data collected from numerous simulation models, Arch2End treats the device inference process as a black-box function that outputs inference latency, thus achieving feature dimensionality reduction through linear fitting. The fitted parameter vector becomes the device's black-box modeling feature. Finally, the unified device modeling vector includes architecture-aware and black-box modeling features.

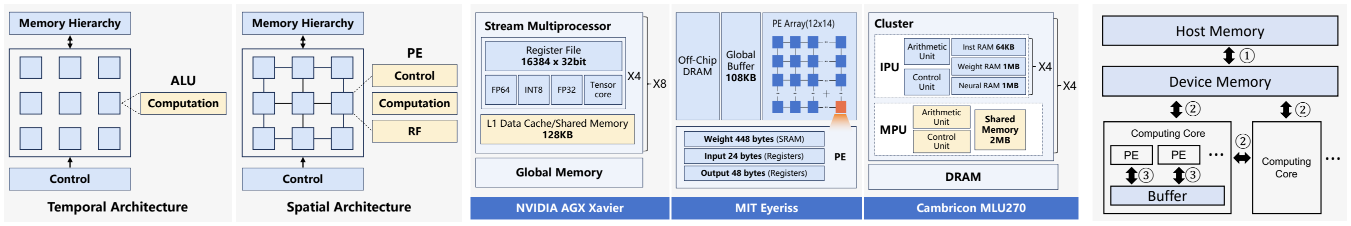
B. Stage one: Architecture-aware benchmark model design

As shown in Fig. 6, differences among intelligent devices are primarily manifested in three dimensions: computation architecture, memory hierarchy, and data communication [25]. First, different computational architectures in Fig. 6(a) offer varied acceleration benefits for operators, which is the most critical factor in device heterogeneity. Moreover, varying memory hierarchies in Fig. 6(b) affect the data access patterns during inference, leading to variations in device performance. Lastly, data communication in Fig. 6(c) influences the efficiency of data exchange between different memory levels during inference, which is closely related to device inference latency. Therefore, this stage focuses on constructing an architecture-aware benchmark model set based on public information, delineating the performance disparities of devices under boundary case conditions. Taking the ten devices mentioned in Section IV-A as examples, Table II lists cases of benchmark models designed from three dimensions.

1) Computation Architecture: Intelligent devices can be broadly categorized into Temporal Architecture and Spatial Architecture [25], as shown in Fig. 6(a).

Devices based on Temporal Architecture, such as CPUs and GPUs, are typically designed using Single-Instruction Multiple-Data (SIMD) or Single-Instruction Multiple-Threads (SIMT) parallel computing models. Multiple computational units can execute a single instruction in parallel. An external scheduler centrally controls these units and exchanges data mainly through shared memory. Temporal Architecture devices offer computational flexibility while accelerating inference.

Representatives of devices based on Spatial Architecture include Google's TPU and MIT's Eyeriss, which often employ multiple homogeneous Processing Elements (PEs) in a spatial array. PEs are interconnected via the Network on Chip (NoC) and have independent computation, Register File (RF), and control components. This setup facilitates data transfer between PEs and supports various dataflow patterns to reuse



(a) Computation Architecture (b) Memory Hierarchy (c) Data Communication

Fig. 6. Different architecture cases for heterogeneous devices in three dimensions: (a) computation, (b) memory, and (c) communication.

intermediate data in computations, such as weight-stationary, output-stationary, and row-stationary. Spatial architecture devices are more efficient for matrix multiplication and addition due to their data flow design, but they lack efficient support for nonlinear functions and complex irregular operations.

Furthermore, many commercial intelligent devices, such as Cambricon MLU and Huawei Ascend 910, integrate the advantages of both architectures through a hybrid architectural design approach. Based on the analysis of computation architectures presented, we propose designing benchmark models for computation architecture from three perspectives.

- **Architectural affinity benchmark** is devised to delineate the distinct advantages and disadvantages of the two architectures. Considering the differing sensitivities of these architectures to data reuse patterns in convolution operations [10], this benchmark can be constructed with convolutional operators that vary in data dimensions.
- **General computational benchmark** aims to evaluate the quantity and capability disparities of general computing units using the most common computational tasks. Given that the current demand for computational power predominantly arises from convolution and matrix multiplication, this benchmark may involve convolutional layers and matrix multiplication tasks of varying computational scales.
- **Special computational benchmark** measures the computational power of specialized device components by designing nonlinear and sparse cases. Many intelligent devices have incorporated support for unique computation modes, such as exponential operations, max operations, and sparse computations. Lack of support for these operations on devices can result in offloading to the CPU, significantly impacting device performance [28].

2) Memory Hierarchy: Fig. 6(b) illustrates the memory hierarchy in intelligent computing devices, with the NVIDIA AGX Xavier, MIT Eyeriss, and Cambricon MLU270 serving as exemplars of temporal, spatial, and hybrid architecture devices, respectively. AGX consists of eight Streaming Multiprocessors (SMs) that share data via the Global Memory. Each SM incorporates four processing blocks, which share data through shared memory and are capable of flexible logical partitioning according to task requirements. Eyeriss connects to off-chip storage through a monolithic design. Its on-chip PE array temporarily stores and shares data through a global buffer, with each PE independently housing on-chip scratchpad memory for inputs, weights, and outputs. MLU270 comprises four *Clusters* and off-chip memory. Each *Cluster* comprises four Intelligent Processing Units (IPUs) and one Memory Processing Unit (MPU), with the MPU facilitating data exchange between IPUs and across *Clusters*.

The data analysis from Fig. 6(b) reveals that intelligent devices typically feature multi-level memory hierarchies with varying capacities and functionalities. The capacity differences across memory levels lead to varied preferences for tasks with different memory demands. For instance, the AGX and Eyeriss, with their numerous smaller registers, offer better performance for small convolution tasks. In contrast, the MLU270, equipped with larger buffer capacities, provides enhanced support for large convolution tasks with greater reuse distances. Beyond memory capacity differences, the division proportion of buffers across different devices also varies, resulting in distinct task affinities. Devices with larger activation buffers are better suited for tasks with larger feature maps, while those with larger weight buffers are more favourable for tasks with larger parameter sizes. Thus, benchmark models are designed from two perspectives:

- **Memory scale awareness benchmark:** Models with varying magnitudes of total memory sizes are designed to probe the memory boundaries of different devices and illustrate how task execution varies with memory size.
- **Memory distribution awareness benchmark:** Under equal total memory sizes, models with different memory distributions (e.g., larger input activations, larger weight parameters) are designed to reflect the functional differences in memory across devices distinctively.

3) Data Communication: Fig. 6(c) illustrates that the data communication impacting DNN inference latency in intelligent devices primarily consists of three parts: ① loading DNN and data, ② communication between computing cores during parallel computing, and ③ intra-core communication.

Before task execution, DNNs and data must be loaded from the host memory to the device memory. Ideally, the communication process for input data overlaps with the neural network's inference process in a pipelined fashion. However, the relative matching of computation and communication capabilities across different intelligent devices may cause overall computational latency to be constrained by the data transmission rate. During task execution, most devices perform model or data parallelism based on the number of computing cores. For instance, in the Cambricon MLU270, the neural network is replicated across different *Clusters* for data parallelism to enhance inference throughput. Such inter-core communication incurs significant communication costs due to the need for network replication and frequent data exchanges. Intra-core communication mainly arises from the execution process of

TABLE II
THE CASE OF ARCHITECTURE-AWARE BASELINE MODEL DESIGN WITH EXPERIMENTAL DEVICES AS AN EXAMPLE.

Dimension	Benchmark Type	Scale[1] ($\mathbb{S}$)
Computation Architecture	Architectural affinity benchmark	Under the premise of keeping one or two values among CI, CO, KH, KW larger while reducing the others, ten different convolution layers are derived, such as [1024, 1, 3, 3, 8, 8].
	General computational benchmark	1) Convolution layers [3/64/256/..., 64/256/1024/..., 3, 3, 56, 56]; 2) Matrix multiplication [50/500/5000/..., 50/500/5000/..., 50/500/5000/...].
	Special computational benchmark	1) Similar to the general component computing power benchmark, the sparsity ratio is set to 50%, 70%, 90%; 2) Vectors of length 5000 are activated by functions such as Sigmoid, ReLU, Tanh, Softmax.
Memory Hierarchy	Memory scale awareness benchmark	Construct convolution layers with large CI, CO, KH, KW, e.g., [64/128/..., 64/128/..., 3, 3, 112/224/..., 112/224/...].
	Memory distribution awareness benchmark	While maintaining the same total storage consumption as the memory scale awareness benchmark, vary parameters to construct different input, output, and weight ratios, e.g., [64, 128, 3, 3, 112, 224], [128, 64, 3, 3, 112, 224].
Data Communication	Host-to-device communication benchmark	Utilize MobileNet series networks, setting image sizes to 56x56, 224x224, 1024x1024, etc.
	Inter-core communication benchmark	Employ ResNet and VGG series networks, configuring various batch sizes, e.g., 1, 4, 16, 64, 128, etc.
	Intra-core communication benchmark	1) Convolution + batch normalization + ReLU; 2) Convolution + ReLU; 3) Convolution + ReLU + Max-pooling; 4) Residual block; 5) Transformer encoder module; 6) Transformer decoder module.

[1] [CI, CO, KH, KW, FH, FW] respectively represent the number of input channels, output channels, kernel height, kernel width, input feature map height, and input feature map width for a convolution layer; [i, j, k] denote the matrix multiplication operation between the $i \times j$ matrix and the $j \times k$ matrix.

operators within the neural network, including the process of writing results back to memory and loading data for the next operation. For DNN tasks with strong inter-layer data dependencies, many approaches fuse multiple layers into a single operator to avoid intermediate result write-back [29]. However, due to differences in hardware architecture, compiler, and deep learning framework support, there is considerable variation in layer fusion and operator implementation across devices.

Thus, following the DNN execution process, we design benchmark models from the following three perspectives.

- **Host-to-device communication benchmark** focuses on the efficiency of data communication across the host and device interface (e.g., PCIe), employing DNN with shorter latencies to emphasize the impact of data transmission on overall latency.
- **Inter-core communication benchmark** evaluates communication efficiency by adjusting the DNNs' batch size. Different batch sizes can influence parallelism strategies (e.g., model and data parallelism), affecting the communication cost during inference [30].
- **Intra-core communication benchmark** consists of operators composed of numerous fusible layers, aiming to reflect intra-core communication performance from the perspective of layer fusion operator implementation.

4) Benchmark model design: Based on the preceding analysis, an architecture-aware benchmark model collection ($\mathbb{S}$) can be designed using publicly available information on heterogeneous devices. Taking the ten devices mentioned in Section IV-A as examples, Table II presents the cases of benchmark model designs for three dimensions. The measured inference latencies of the benchmark models can serve as **architecture-aware features** of the devices, which can be described as:

$$HP_{h_k} = [p_{h_k,x_1}, p_{h_k,x_2}, \dots, p_{h_k,x_m}] \quad (h_k \in H_\xi, x_i \in \mathbb{S}), \tag{1}$$

where, p_{h_k,x_i} is the inference latency of the models (x_i) on the platform (h_k), x_i is the sample of the architecture-aware benchmark model set ($\mathbb{S}$). Moreover, as the number of heterogeneous devices grows and more information is disclosed, $\mathbb{S}$ can be expanded by exploring the architecture information, thus showcasing the scalability of this method.

C. Stage two: Black-box modeling of inference latencies

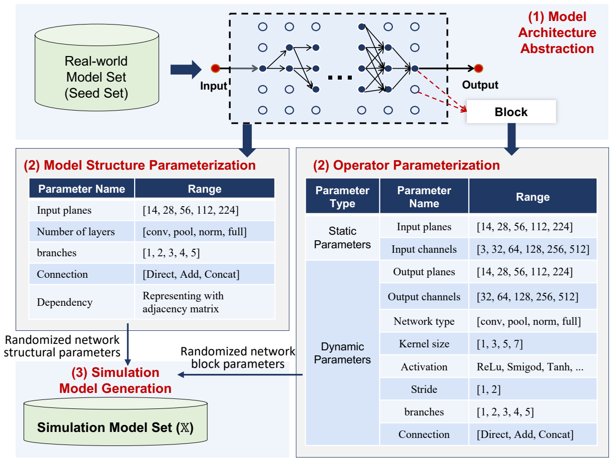
Since stage one can only design a limited set of benchmark models based on publicly available architecture information, it inevitably falls short of covering device diversity adequately. To compensate for the benchmark models' limited capability in perceiving device performance, stage two introduces a black-box analysis method based on end-to-end inference metrics, primarily comprising simulation model generation via generators and device feature dimensionality reduction through linear fitting, as shown in Fig. 4. Specifically, this stage generates extensive simulation models through a model generator and deploys them on different devices to collect model-latency data. However, the abundance of simulation models may include highly similar structures or models that cannot be inferred on some devices, leading to data pair redundancy and gaps. To characterize device performance with more refined vectors, we propose a feature dimensionality reduction strategy based on linear fitting, mapping numerous data pairs to unified fitting parameters. The normalized fitting parameters are the device's black-box modeling vector, enabling comprehensive and unified device characterization.

Next, we focus on constructing the simulation model generator and the feature dimensionality reduction strategy.

1) Simulation model generator: Fig. 7 demonstrates the three steps of the simulation model generator realization.

Network Architecture Abstraction. To simplify the construction of simulation models, the generator abstracts the continuous operator set with complex data dependencies in the seed model into blocks, serving as the basic units for simulation model design. It facilitates the fusion of different model structures to generate the simulation model. Therefore, variations in the operator set and model structure rules are crucial for the quality of case generation, achievable through the extraction of operator parameter spaces and model structure parameter spaces from the seed models.

Parametrization. Operator parametrization guides the generation of blocks, including both static and dynamic param-


Real-world Model Set (Seed Set)
Input
Output
Block
(1) Model Architecture Abstraction
(2) Model Structure Parameterization
(2) Operator Parameterization
Randomized network structural parameters
Randomized network block parameters
(3) Simulation Model Generation
Simulation Model Set ($\mathbb{X}$)

Parameter Name	Range
Input planes	[14, 28, 56, 112, 224]
Number of layers	[conv, pool, norm, full]
branches	[1, 2, 3, 4, 5]
Connection	[Direct, Add, Concat]
Dependency	Representing with adjacency matrix

Parameter Type	Parameter Name	Range
Static Parameters	Input planes	[14, 28, 56, 112, 224]
	Input channels	[3, 32, 64, 128, 256, 512]
Dynamic Parameters	Output planes	[14, 28, 56, 112, 224]
	Output channels	[32, 64, 128, 256, 512]
	Network type	[conv, pool, norm, full]
	Kernel size	[1, 3, 5, 7]
	Activation	ReLu, Smigod, Tanh, ...
	Stride	[1, 2]
	branches	[1, 2, 3, 4, 5]
	Connection	[Direct, Add, Concat]

Fig. 7. Simulation Model Generator. (1) The generator abstracts the parameter types of operators and model structures from a seed set. (2) Ranges for parameters are customized based on the types and dependencies of blocks in real-world models. (3) Using randomly generated parameters, blocks are integrated into the overall model structure to form a simulation model.

eters. Static parameters, determined by the previous layer, encompass the plane and channel of input feature maps; dynamic parameters are randomly generated at runtime within specific ranges, including the output feature map plane and channel, layer type, kernel size, activation, stride, branch, and connection, ensuring diversity in operator types and connections within blocks. Model structure parametrization guides the overall model structure definition, such as single-branch, multi-branch, dual, and residual structures, parametrized by input feature map plane, number of layers, branch numbers per layer, and interlayer data dependencies.

Model Generation. After parametrizing operators and model structures, determining the value range for the altered parameters is essential. The generator identifies the value boundaries of various parameters within the seed models, then covers the design space with a value range slightly larger than it, as shown in the tables in Fig. 7. Based on above parameter space, the generator randomly varies model structure and network block parameters, outputting the final simulation model cases. **The inference execution of numerous simulation models involves both the software stack and hardware, with their end-to-end latency reflecting the impact of software-hardware collaboration on device performance.**

2) Device feature dimensionality reduction strategy: Due to the randomness, simulation models may be instances of high similarity or inability to execute on some devices, leading to redundancy and gaps in model-latency data pairs. This section introduces a linear fitting-based feature dimensionality reduction strategy to extract critical features related to device capabilities from a large volume of data pairs. This strategy begins with acquiring a vast collection of model-latency data pairs via the simulation model generator, followed by extracting key model features based on *Pearson coefficients* [31] of the parametric features shown in Fig. 7; and finally, performing linear fitting on the model-latency data pairs using the extracted model key model features as independent variables, from which fitting parameters are derived as the unified device

Algorithm 1 Key Model Feature Extraction

```
Input: $F$: model features vector; $T$: latency; $K$: max-length of $V$; $E$: linear
    fitting error; $\varepsilon$: min-threshold of $E$.
Output: $V$: key model features
 1: function KMFE($F$, $T$)
 2:     Load $F$ and $T$, Initialize $V$ by NULL
 3:     while length of $V$ less than $K$ do
 4:         Calculate Pearson coefficient ($P$) of features in $F$ w.r.t. $T$
 5:         Select the feature with the largest $P$ into $V$
 6:         Delete the selected feature from $F$
 7:         Calculate $E$ of each feature in $V$ after linear fitting
 8:         if $E < \varepsilon$ then
 9:             break
10:         end if
11:         $T \leftarrow E$
12:     end while
13:     return $V$
14: end function
```

black-box modeling features. The dimensionality reduction process is described in detail next.

Initially, we generate a set of simulation models and obtain their inference latencies on each device as follows:

$$Sim_{h_k} = [y_{h_k,x_1}, y_{h_k,x_2}, \dots, y_{h_k,x_d}] \quad (h_k \in H_\xi), \tag{2}$$

where y_{h_k,x_i} is the inference latency of the model (x_i) on the device (h_k), x_i is the sample of the simulation models ($\mathbb{X}$), fixed across all tasks for each device. For the size (d) of $\mathbb{X}$, we customize it based on the size of the seed model set and randomly select d simulation models from the generator to represent the complete DNN set.

Subsequently, the key model feature (V) extraction process based on *Pearson coefficients* [31] primarily identifies the model features most correlated with inference latency from the parametric feature (F) depicted in Fig. 7. This process reduces the feature dimensionality by eliminating irrelevant features, decreasing the likelihood of overfitting downstream tasks. Specifically, we select the model features with the highest correlation coefficients for inclusion in the key feature set V and continue extracting features using the linear residual fitting method until the number of key model features reaches a predetermined threshold (K), or the fitting error falls below the min-threshold (ε). The procedure of extracting the key model features is illustrated in the Algorithm 1.

The feature dimensionality reduction process extracts the device's black-box modeling features by fitting the relationship between the feature set V and inference latencies y. Specifically, a linear fitting function ($\mathbb{F}_{h_k}(\cdot)$) is trained on a single device, with V_{x_i} as the input and y_{h_k,x_i} as the output. The device's black-box modeling features are the normalized fitting parameters (w_{h_k}) corresponding to each key model feature. The formal expression of the linear fitting function is as follows:

$$\begin{aligned} \mathbb{F}_{h_k}(V_{x_i}; w_{h_k}) = & V_{x_i} \times w_{h_k} \to y_{h_k,x_i} \\ & (x_i \in \mathbb{X}, h_k \in H_\xi), \end{aligned} \tag{3}$$

where parameters w_{h_k} is used for the linear fit of V_{x_i} to y_{h_k,x_i}.

Finally, the **black-box modeling features** of the device are represented by the parameter vector of the linear fitting function as follows:

$$HE_{h_k} = [w_{h_k,1}, w_{h_k,2}, \dots, w_{h_k,n}] \quad (h_k \in H_\xi). \tag{4}$$

D. Unified Modeling Result

The two stages offer distinct focal points for intelligent device modeling. Stage one is devoted to designing architecture-aware benchmark model sets from computational, memory, and communication dimensions using limited architecture information, employing benchmark model inference latency as coarse-grained features for architecture awareness. Stage two extracts device performance-related features from extensive simulation model latency data through black-box analysis, enabling comprehensive modeling analysis without needing in-depth hardware architecture dissection. The unified device modeling vector of Arch2End represents a combination of features from both stages, summarized as follows:

$$\begin{aligned} HA_{h_k} = [HP_{h_k}, HE_{h_k}] = &[p_{h_k,x_1}, p_{h_k,x_2}, \ldots, p_{h_k,x_m}, \\ &w_{h_k,1}, w_{h_k,2}, \ldots, w_{h_k,n}] \quad (h_k \in H_\xi). \end{aligned} \tag{5}$$

Modeling Applications. HA_{h_k} facilitates a quantitative evaluation of heterogeneous device performance. HP_{h_k} represents the benchmark model's inference performance, serving as a direct metric for quantifying device performance. HE_{h_k}, derived from end-to-end inference latencies, offers a comprehensive characterization of device performance within the standardized vector. In practical applications, Arch2End proves highly versatile. For instance, in predicting DNN inference latency across heterogeneous devices, HA_{h_k} delivers an all-encompassing performance overview of the device, which is incorporated into the prediction model as a device feature vector. For device grouping, HP_{h_k} acts as a quantitative performance indicator, contributing to the computational assessment of device groups to maintain grouping result balance.

IV. Experiment and Evaluation

This section provides two case studies demonstrating Arch2End's practical application in intelligent scenarios. **Case 1** utilizes Arch2End to predict DNN inference latency across various datasets, including NAS-Bench-201 and real-world DNNs. The prediction error is assessed via *Absolute Percentage Error* (APE), i.e., $|\frac{A_t - P_t}{A_t}|$, where A_t and P_t denote the actual and predicted values at data point t, respectively. Additionally, accuracy metrics of 5%, 10%, and 15% are used to gauge the proportion of models whose prediction *APE* falls within these ranges. **Case 2** explores inter-group balanced device grouping, aiming to demonstrate the advantages of Arch2End in device grouping. Specifically, devices are grouped based on modeling features, and numerous random task loads are constructed to test the execution latency of each group. The *standard deviation* in execution latency serves as a metric to assess the balance among device groups.

A. Experiment Setup

1) Heterogeneous Device Platforms: In order to realize a highly heterogeneous scenario, the devices in this experimental environment include a variety of GPUs and NPUs, whose architectural specifications are shown in the public datasheets on the official website [32], [33]. The GPUs include *NVIDIA P100, NVIDIA V100, NVIDIA TITAN XP, NVIDIA P4, Jetson Xavier AGX, Jetson Xavier NX,* and *Jetson TX2*. The NPUs include *Cambricon MLU220* and *Cambricon MLU270. MIT Eyeriss*, implemented by the Timeloop simulator [9], is added to ensure the completeness of the device types.

2) Datasets: The experiment datasets comprise the NAS-Bench-201 dataset [26] and the real-world DNNs. NAS-Bench-201, a public dataset of diverse neural networks, evaluates the representational capabilities of heterogeneous device modeling through multi-device latency prediction accuracy. The real-world DNNs includes *DenseNet121, DenseNet169, EfficientNetB0, EfficientNetB1, EfficientNetB2, ResNet50, ResNet101, VGG11, VGG16, EdgeViT-XS, EfficientViT-M1* and *MobileViT-XS*, employed to assess the prediction accuracy of real-world DNN inference latency.

3) Comparison Baselines: To better evaluate the contribution of each stage to device representation, we designed ablation experiments for the two-stage modeling of Arch2End. In this setup, Arch2End-HP utilizes only the inference latency vector from the benchmark models as the modeling outcome, denoted as *HA=HP*. Arch2End-HE employs solely the standardized parameter vector extracted from black-box modeling, denoted as *HA=HE*. Arch2End combines vectors from both stages as the modeling result, indicated as *HA=[HP, HE]*.

As a unified modeling method for heterogeneous devices, Arch2End's comparison baselines are also tailored for devices with diverse architecture and configuration types. Accordingly, we establish six comparison baselines: 1) *ONE-HOT* encoding is considered for device representation without modeling, expanding with the addition of device types. 2) *FLOPS* [13] and *PCN* [16] are utilized for modeling based on public architecture information, where *FLOPS* uses peak computational capabilities, and *PCN* employs generic architecture configurations for devices. 3) End-to-end inference data-based modeling methods include *MLPerf* [20] and *HELP* [22]. MLPerf uses model inference latency vectors from its benchmark suite to depict heterogeneous platforms. *HELP* enhances the suite by collecting runtime model inference latencies to improve device performance representation. 4) *PCN+HELP* is considered a method that combines optimal architecture information integration and end-to-end inference metric analysis.

B. Case 1: Latency Prediction on Heterogeneous Devices

Case 1 evaluates Arch2End's capability to predict neural network latency, covering NAS models and real-world DNNs. The prediction experiment is shown in Fig. 8, which presents the Latency Prediction Model (LPM) incorporating the self-attention and cross-attention mechanism. In model evaluation, the data are randomly split into training and validation sets in an 8:2 ratio, with the data re-partitioned in each of the 20 repeated training cycles. The training consistently employs a fixed learning rate (0.001), optimizer (Adam), batch size (600), and epoch count (150). After training, the model's prediction error and confidence are evaluated on the validation set, with repeated experiments assessing the average accuracy and its standard deviation across different relative error ranges.

1) NAS Model Prediction: Table III presents the DNN latency prediction percentage errors on the NAS-Bench-201 dataset for ten devices simultaneously. The average *APE* of

TABLE III
MEAN PREDICTION ACCURACY OF NAS-BENCH-201 WITHIN DIFFERENT APE RANGE ON TEN TYPES OF DEVICES.

APE Range	**Prediction Accuracy (%) ± Standard Deviation (10^{-2})**								
	Not modeling	Architecture information		End to end inference metrics		Combination	**Ours**		
	ONE-HOT	FLOPS	PCN	MLPerf	HELP	PCN+HELP	Arch2End-HP	Arch2End-HE	**Arch2End**
5%	44.34±3.22	40.78±1.92	55.22±8.78	51.41±6.12	57.44±5.07	64.6±3.36	73.37±2.78	76.67±3.51	**94.31±0.82**
10%	72.16±3.64	68.79±2.00	77.9±8.77	79.32±4.49	84.98±3.49	86.45±3.22	87.85±1.44	92.62±1.48	**98.84±0.24**
15%	85.05±2.55	82.5±1.38	88.19±7.00	89.63±2.81	93.99±1.96	94.71±2.17	92.59±0.86	96.3±0.83	**99.53±0.08**

TABLE IV
APE OF DIFFERENT MODELING METHODS FOR REAL-WORLD DNN LATENCY PREDICTION.

Method	**Absolute Percentage Error (APE, %) ± Standard Deviation**										
	P100	**V100**	**TITAN XP**	**P4**	**AGX**	**NX**	**TX2**	**MLU220**	**MLU270**	**Eyeriss**	**Average**
PCN	10.12±15.35	26.56±23.55	14.66±22.77	15.41±12.72	19.87±17.84	39.91±23.61	43.28±13.16	36.34±28.42	44.25±16.54	15.29±24.68	26.57
HELP	26.85±22.15	14.16±14.98	32.56±20.45	13.95±14.34	18.24±17.48	12.36±10.08	14.29±20.82	35.15±28.16	30.53±29.26	13.38±16.82	21.15
PCN+HELP	19.37±13.82	14.57±13.16	20.26±15.64	11.64±15.04	6.57±14.69	10.77±15.57	14.73±16.4	33.69±29.27	36.44±29.95	11.85±15.63	17.99
Arch2End	11.25±8.77	6.63±3.89	6.52±5.46	6.7±6.13	5.58±4.15	8.19±4.48	10.46±7.93	9.02±5.5	9.71±5.71	7.24±4.07	8.13

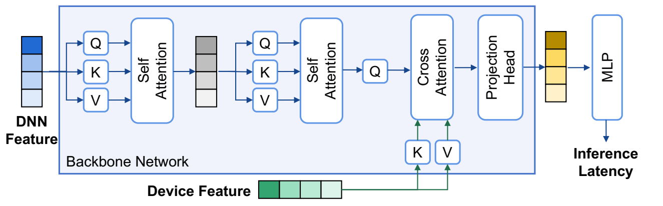


Fig. 8. Latency Prediction Model (LPM). Its inputs include DNN features (NAS-Bench-201's encoded vectors) and device features (results from different device modeling methods). Its output is the predicted inference latency.

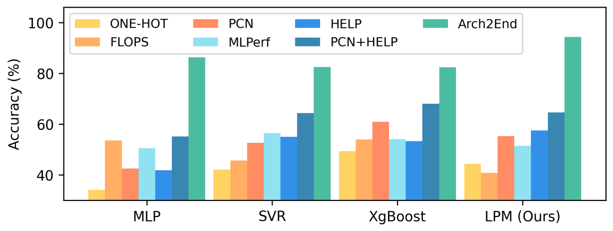


Fig. 9. Mean prediction accuracy of different prediction models within 5% APE range. The inputs and outputs of the mentioned prediction models are consistent. During training, the relevant settings for MLP are the same as for LPM. SVR settings include kernel='linear', C=100, gamma=0.1, epsilon=0.1. XGBoost settings include n-estimators=15, learning rate=0.1, max depth=2.

the prediction results of Arch2End is 1.7%, and the prediction accuracies at the 5% and 10% *APE* ranges are 94.3% and 98.8%, respectively. Meanwhile, the prediction accuracies using architecture-aware features or black-box modeling features decrease slightly but remain above 92%.

As shown in Table III, Arch2End's modeling methods (Arch2End-HP 87.85%, Arch2End-HE 92.62%, Arch2End 98.84%) significantly outperform the baseline models (ONE-HOT 72.16%, FLOPS 68.79%, PCN 77.9%, MLPerf 79.32%, HELP 84.98%, PCN+HELP 86.45%) in prediction accuracy within a 10% *APE* range. Notably, Arch2End shows the best stability with standard deviation within 0.82.

ONE-HOT, used for comparative analysis, lacks device modeling, resulting in poor predictions. FLOPS, identifying devices by peak computational capacity, and PCN, relying on opaque architecture information, fail to adequately capture performance variances among increasingly heterogeneous devices. Similarly, limited reliance on end-to-end inference latency data (MLPerf and HELP) fails to accurately reflect device performance across varied DNNs. Arch2End, by effectively integrating both types of features, improved accuracy within the 10% *APE* range by 12.39% with a standard deviation of only 0.24 compared to PCN+HELP.

Furthermore, to validate the robustness of Arch2End on other prediction models, we added common machine learning models such as *MLP*, *SVR*, and *XGBoost* [34]. Fig.9 displays the prediction outcomes under various prediction models and modeling methods combinations. Among the four predictive models, Arch2End performs the best, achieving an absolute improvement in accuracy of 14.4% to 31.2% over the second-best model (PCN+HELP).

2) Real-world DNN Prediction: To demonstrate the advantages of Arch2End in real-world DNN latency prediction, Table IV compares the predictions of various modeling methods, and Table V shows Arch2End's prediction result in detail. The *APE* of the real-world DNNs is about 8.13%, which is significantly lower than that of PCN (26.57%), HELP (21.15%), and PCN+HELP (17.99%) with the same devices and tasks. The *APE* of HELP and PCN+HELP on *EdgeViT, EfficientViT* and *MobileViT* reaches more than 34% due to the lack of generalization to new network structures and operators, especially on NPUs. PCN has poor prediction accuracy and stability in most models due to the lack of consideration for diverse DNNs. In comparison, Arch2End not only perceives the overall performance boundaries of devices through benchmark models but also covers a wide range of network architectures and operators via the simulation model generator. Consequently, Arch2End outperforms other modeling methods in terms of *APE* and stability in prediction.

3) Prediction Error Analysis: The errors in Arch2End primarily originate from two sources: (1) Small models, such as *EfficientNetB0*, *VGG11*, and *MobileViT XS*, typically exhibit greater relative errors due to shorter inference times, with

TABLE V
APE OF ARCH2END IN REAL-WORLD DNN LATENCY PREDICTION.

network	Arch2End APE(%) of Each Device									
	P100	V100	TITAN XP	P4	AGX	NX	TX2	MLU 220	MLU 270	Eyeriss
DenseNet121	14.40	3.08	2.75	5.99	0.60	4.11	3.71	1.12	4.61	1.85
DenseNet169	27.39	3.36	13.39	1.90	3.86	3.74	8.23	9.46	8.55	9.83
EfficientNetB0	8.84	8.42	4.05	4.75	13.70	15.38	24.48	3.58	12.65	12.39
EfficientNetB1	8.20	2.76	5.07	7.25	4.77	6.58	2.69	10.15	0.87	11.28
EfficientNetB2	5.05	5.58	1.37	4.25	1.36	0.52	1.77	13.21	5.16	4.70
ResNet50	6.81	7.54	6.93	11.74	10.11	10.59	16.53	4.96	13.12	11.62
ResNet101	1.66	7.83	10.44	4.98	6.30	7.59	4.13	18.38	17.28	2.09
VGG11	26.49	15.10	1.95	0.13	9.57	14.17	16.60	12.62	15.90	1.99
VGG16	7.06	5.48	2.64	4.04	5.30	7.64	8.67	13.58	15.38	7.98
EdgeViT_XS	6.58	7.13	16.65	21.96	0.23	11.57	17.83	3.15	3.60	8.72
EfficientViT_M1	1.27	2.25	0.24	14.86	2.09	6.64	7.18	4.50	5.70	5.01
MobileViT_XS	17.59	12.06	12.68	10.42	4.73	3.90	21.92	-[1]	-[1]	0.61
Average	**11.25**	**6.63**	**6.52**	**6.70**	**5.58**	**8.19**	**10.46**	**9.02**	**9.71**	**7.24**

[1] The inference testing of MobileViT-XS is unavailable due to the lack of support for operators with large feature maps on MLU270 and MLU220.

latencies under 150ms. For these models, even minor absolute errors can result in significant relative errors. This type of error is particularly pronounced with other modeling methods. Compared to PCN+HELP, Arch2End significantly reduces the average relative error for the small models in Table V from 28.3% to 10.9%. (2) Models with long dependencies, such as *DenseNets*, require larger buffer capacities to store feature maps from other layers, minimizing frequent data exchanges between buffers and memory that degrade performance. This impact is most critical on the P100 because it has the smallest shared memory/L1 cache (24K) among GPU devices. Additionally, the effect becomes more noticeable with increased network layers, as observed with *DenseNet169* on the P100.

C. Case 2: Inter-group Balanced Device Grouping

Case 2 focuses on applying Arch2End results to inter-group balanced device grouping. This process entails evenly grouping devices based on various modeling features to ensure similar computation task completion times, minimizing the *standard deviation* in execution latency. Due to the diversity and randomness of actual tasks, the task sets executed by each device group follow the *Poisson Distribution*.

As shown in Section IV-A, device features from the mentioned modeling methods are represented by vectors composed of multiple feature values, collectively denoted as U. In Arch2End, this is expressed as $U = HA$. Since each feature value denotes the device's computational capability, the feature set of a device group can be defined as the summation of corresponding feature values, i.e., $F_{group_i,k} = \sum_{j \in group_i} U_{j,k}$. Where, $F_{group_i,k}$ represents the k^{th} feature value of group i, $j \in group_i$ stands for iterating over each device within the $group_i$, and $U_{j,k}$ denotes the k^{th} feature value of device j.

To achieve balanced grouping across all feature values, the grouping process incorporates a local search heuristic algorithm [35], with the specific objective formalized as follows:

$$\min Var_h = \sum_{k \in U} Var(F_{group_i,k} | group_i \in G), \tag{6}$$

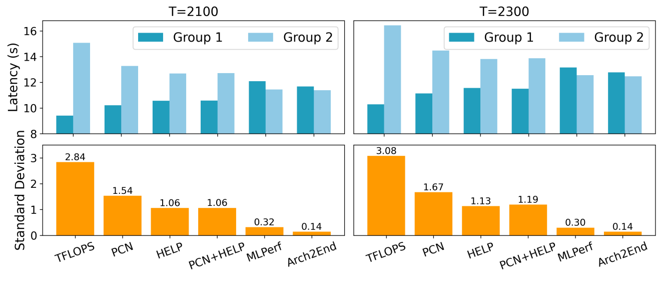


Fig. 10. Evaluation in the two-group experiment. The evaluation metrics for each method consisted of the execution latency of each group and the standard deviation between groups.

TABLE VI
STANDARD DEVIATION OF GROUPS IN FOUR-GROUP EXPERIMENTS.

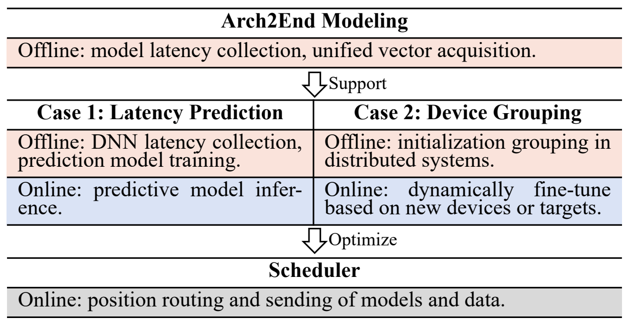
	T=1900	T=2100	T=2300	T=2500
FLOPS	1.693	1.882	2.046	2.216
PCN	1.078	1.192	1.305	1.412
HELP	0.886	0.972	1.072	1.152
PCN+HELP	1.336	1.468	1.612	1.722
MLPerf	0.438	0.505	0.548	0.584
Arch2End	**0.118**	**0.130**	**0.126**	**0.130**

where, Var_h denotes the inter-group performance variance metric, U represents the set of device features, and G is the set of device groups.

This section evaluates the device group load by the latencies incurred while executing the same task set. The standard deviation in latencies across groups is used to quantify the degree of inter-group balance offered by various modeling methods. Device grouping aims to organize and schedule devices in overloaded clusters to distribute the load evenly across groups. Hence, reflecting the performance review of the device groups in the experiment, we deliberately choose *the task arrival rate per second* (T) to exceed 1900, pushing the groups into an overloaded state. T manipulates device loads to assess their equilibrium under diverse workloads. The experiment specifies a batch size (16) for inference images. The duration of each experiment is 10 seconds, and the grouped devices include nine types of GPUs and NPUs, detailed in Section IV-A.

The results of the two-group experiment on 27 devices are shown in Fig. 10. There are nine types of devices, including GPUs and NPUs, three of each device. For the two groups of devices divided by Arch2End, the latencies for completing the same set of tasks are much closer to each other, with standard deviations within 0.14 ($T = 2100$) and 0.14 ($T = 2300$). At $T = 2300$, the latency variance between groups for Arch2End decreases by 95.4%, 91.5%, 87.5%, 88.1%, and 52.8% for FLOPS, PCN, HELP, PCN+HELP, and MLPerf, respectively.

To further validate the advantages of Arch2End, we extended the four-group experiment on 54 devices and under different loads. There are nine types of devices, including GPUs and NPUs, six of each device. As shown in Table VI, the standard deviation of groups for Arch2End is significantly lower than the other five baseline methods. As the value of T increases, the load standard deviation of FLOPS, PCN, HELP, PCN+HELP, and MLPerf increases rapidly. Arch2End has outstanding stability with varying loads, with less than 0.012 fluctuations in latency standard deviation between groups.



Arch2End Modeling

Offline: model latency collection, unified vector acquisition.

Support

Case 1: Latency Prediction	**Case 2: Device Grouping**
Offline: DNN latency collection, prediction model training.	Offline: initialization grouping in distributed systems.
Online: predictive model inference.	Online: dynamically fine-tune based on new devices or targets.

Optimize

Scheduler

Online: position routing and sending of models and data.

Fig. 11. Cost analysis including device modeling, cases and scheduling.

D. Cost Analysis

The cost analysis comprises three stages, as shown in Fig. 11. The cost of **Arch2End modeling** occurs solely during the offline phase, including: 1) Inference latency collection for benchmark and simulation models, dependent on the model count and device performance, with the highest (V100) and lowest (TX2) performance devices requiring 0.5 and 6.3 hours respectively in our experiments. 2) The fitting cost for the unified parameter vector in Stage 2 is minimal for linear fitting small data batches and thus negligible.

In **Case 1**, the offline cost for DNN latency prediction involves collecting latencies and training the prediction model. The online phase allows for rapid latency prediction through batch input of task and device features, completing the prediction of 5,400 latencies for 200 DNNs and 27 devices in 0.1 seconds.

In the ideal scenario of no waiting and concurrent testing between devices, the total time cost is primarily constrained by the processing speed of the slowest device, such as the TX2, which may exceed 450s. Furthermore, as the prediction cost constitutes a small portion of the overall execution latency, it can be offset through pipeline parallelism in continuous tasks [36]. In **Case 2**, device grouping costs depend on the replaceable grouping strategy, such as the heuristic search [35]. Offline costs occur only during the initialization process of the distributed system. Subsequently, adding new devices requires only online fine-tuning using group vectors (F_{group}) with a $O(n)$ complexity, where n is the number of groups. Related work on downstream tasks [16], [35] provides a detailed analysis of these costs.

Based on the task scheduling directions determined by latency prediction and device grouping, **scheduling** costs include position routing and sending models and data. This part belongs to the inherent cost of the scheduling algorithm and has been detailed in many scheduling works [5], [6], [16]. Thus, it is not the focus of analysis in this paper.

V. Related Work

A. Scheduling in Embedded Distributed Systems

Embedded distributed systems are tasked with scheduling a wide array of DNNs across heterogeneous devices to enhance system throughput and minimize task wait times [1]. While extensive work utilizes measured DNN latencies for task scheduling [4], [5], [37], [38], the associated costs increase polynomially with task and device quantities. Some work mitigates this by predicting DNN latencies offline [16], [39], [40], but the lack of comprehensive device modeling affects prediction accuracy. With system scale expansion, hierarchical scheduling through device grouping gains importance [17], [35], simplifying scheduling and reducing wait times, yet device heterogeneity complicates unified grouping metrics. These challenges underscore the necessity of a unified modeling framework for heterogeneous devices.

B. Device Modeling Methods

Recently, device modeling methods have been broadly categorized into fine-grained and coarse-grained.

Fine-grained modeling methods analyze hardware architecture and DNN operator details to characterize device performance. Work like Timeloop [9], MAESTRO [10], and nn-Meter [11] investigates architectural differences across devices, focusing on DNN inference acceleration through data mapping, memory reuse, and operator optimization. While these methods provide detailed insights into specific device performance, the proprietary nature of hardware designs limits their generalization across heterogeneous devices [24], [25]. The reliance on extensive architecture information and manual analysis poses challenges for applying these methods.

Coarse-grained modeling methods leverage accessible device data and end-to-end benchmarks to generalize device performance. The Roofline model [18] quantifies computational differences through peak instruction rates but falls short in reflecting the nuances of DNN inference due to fluctuating resource utilization. Work like PCN [16] and Glimpse [19] offers hardware-based predictions for GPU DNN latencies without invasive measures, but the diversity of hardware architectures limits their applicability. Comprehensive benchmarking suites such as MLPerf [20], AIPerf [21], and HELP [22] attempt to bridge these gaps by assessing devices against a selection of real-world DNNs. However, their effectiveness is constrained by the limited scope of the DNNs used, highlighting a challenge in capturing the full spectrum of device performance across varied DNN workloads.

Among all these works on inference scheduling and device modeling, the unique contribution of Arch2End lies in effectively compensating for the shortcomings of architecture information and end-to-end inference metrics through its two-stage modeling process. It provides a unified representation vector for heterogeneous devices, offering more reliable device performance perception for distributed systems.

VI. Conclusion

This paper proposes Arch2End, a two-stage unified system-level modeling method that utilizes a unified feature vector to depict the performance of heterogeneous devices. Abstracting architecture features enables uniform analysis across heterogeneous devices, and benchmark models are designed from three dimensions to probe the boundaries of device performance. The black-box analysis, based on end-to-end inference metrics, not only designs simulation models to expand the coverage of device performance but also maps extensive inference metrics into a unified vector through dimensionality reduction. Experimental results demonstrate the effectiveness of this method in DNN inference latency prediction and device grouping strategy. Future work will integrate the unified modeling method with more intelligent application scenarios to further enhance the performance of heterogeneous distributed systems.

References

[1] Thomas Barnett, Shruti Jain, and et al. Cisco visual networking index (vni) complete forecast update, 2017–2022. *Americas/EMEAR Cisco Knowledge Network (CKN) Presentation*, pages 1–30, 2018.

[2] Yangguang Cui, Kun Cao, Guitao Cao, Meikang Qiu, and Tongquan Wei. Client scheduling and resource management for efficient training in heterogeneous iot-edge federated learning. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(8):2407–2420, 2022.

[3] Matthew Sibanda, Ernest Bhero, and John Agee. Ai edge processing-a review of distributed embedded systems. In *2023 31st Southern African Universities Power Engineering Conference (SAUPEC)*, pages 1–6. IEEE, 2023.

[4] Liekang Zeng, Xu Chen, Zhi Zhou, and et al. Coedge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Trans. Netw.*, 29(2):595–608, 2021.

[5] Wonik Seo, Sanghoon Cha, Yeonjae Kim, and et al. Slo-aware inference scheduler for heterogeneous processors in edge platforms. *ACM Trans. Archit. Code Optim.*, 18(4):43:1–43:26, 2021.

[6] Shuai Liu, Zidong Wang, and et al. Distributed set-membership filtering for multirate systems under the round-robin scheduling over sensor networks. *IEEE Trans. Cybern.*, 50(5):1910–1920, 2020.

[7] Wei Gao, Zhisheng Ye, and et al. Unisched: A unified scheduler for deep learning training jobs with different user demands. *IEEE Trans. Computers*, 73(6):1500–1515, 2024.

[8] Sparsh Mittal and et al. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, 2015.

[9] Angshuman Parashar, Priyanka Raina, and et al. Timeloop: A systematic approach to DNN accelerator evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*, pages 304–315. IEEE, 2019.

[10] Hyoukjun Kwon, Prasanth Chatarasi, and et al. MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. *IEEE Micro*, 40(3):20–29, 2020.

[11] Li Lyna Zhang, Shihao Han, and et al. nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services, Virtual Event, Wisconsin, USA*, pages 81–93. ACM, 2021.

[12] Weihong Liu, Jiawei Geng, Zongwei Zhu, and et al. Ace-sniper: Cloud-edge collaborative scheduling framework with DNN inference latency modeling on heterogeneous devices. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 43(2):534–547, 2024.

[13] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[14] Hanfei Yu, Athirai A. Irissappane, and et al. Faasrank: Learning to schedule functions in serverless platforms. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Washington, DC, USA, September 27 - Oct. 1, 2021*, pages 31–40. IEEE, 2021.

[15] Diana Marculescu, Dimitrios Stamoulis, and Ermao Cai. Hardware-aware machine learning: modeling and optimization. In Iris Bahar, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, page 137. ACM, 2018.

[16] Weihong Liu, Jiawei Geng, and et al. Sniper: cloud-edge collaborative inference scheduling with neural network similarity modeling. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 505–510. ACM, 2022.

[17] Zhijun Wang, Huiyang Li, Zhongwei Li, and et al. Pigeon: an effective distributed, hierarchical datacenter job scheduler. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 246–258. ACM, 2019.

[18] Nan Ding, Muaaz G. Awan, and Samuel Williams. Instruction roofline: An insightful visual performance model for gpus. *Concurr. Comput. Pract. Exp.*, 34(20), 2022.

[19] Byung Hoon Ahn, Sean Kinzer, and et al. Glimpse: mathematical embedding of hardware specification for neural compilation. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 1165–1170. ACM, 2022.

[20] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, and et al. Mlperf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device AI. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022.

[21] Zhixiang Ren, Yongheng Liu, Tianhui Shi, Lei Xie, and et al. Aiperf: Automated machine learning as an AI-HPC benchmark. *Big Data Min. Anal.*, 4(3):208–220, 2021.

[22] Hayeon Lee, Sewoong Lee, and et al. Hardware-adaptive efficient latency prediction for nas via meta-learning. *Advances in Neural Information Processing Systems*, 34:27016–27028, 2021.

[23] David Kanter. Supercomputing 19: Hpc meets machine learning. Website, 2019. https://www.realworldtech.com/sc19-hpc-meets-machine-learning/.

[24] Chen Yiran, Xie Yuan, Song Linghao, Chen Fan, and Tang Tianqi. A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020.

[25] Maurizio Capra, Beatrice Bussolino, and et al. An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet*, 12(7):113, 2020.

[26] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[27] Sungil Kim and Heeyoung Kim. A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting*, 32(3):669–679, 2016.

[28] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, and et al. Full stack optimization of transformer inference: a survey. *CoRR*, abs/2302.14017, 2023.

[29] Jie Zhao, Xiong Gao, Ruijie Xia, and et al. Apollo: Automatic partition-based operator fusion through layer by layer optimization. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022.

[30] Yu Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and CPU platforms for deep learning. *CoRR*, abs/1907.10701, 2019.

[31] Jeremy Adler and Ingela Parmryd. Quantifying colocalization by correlation: The pearson correlation coefficient is superior to the mander's overlap coefficient. *Cytometry Part A*, 77A(8):733–742, 2010.

[32] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on gpus. *J. Syst. Archit.*, 99, 2019.

[33] Tao Luo, Shaoli Liu, Ling Li, and et al. Dadiannao: A neural network supercomputer. *IEEE Trans. Computers*, 66(1):73–88, 2017.

[34] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. Performance prediction for convolutional neural networks in edge devices. *arXiv preprint arXiv:2010.11297*, 2020.

[35] Yubin Duan and Jie Wu. Optimizing resource allocation in pipeline parallelism for distributed DNN training. In *28th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2022, Nanjing, China, January 10-12, 2023*, pages 161–168. IEEE, 2022.

[36] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel CPU/GPU scheduling for multi-dnn real-time inference. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 392–405. IEEE, 2019.

[37] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, and et al. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[38] Jiamin Li, Hong Xu, Yibo Zhu, and et al. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 835–850. ACM, 2023.

[39] Zhihao Jia, James Thomas, Todd Warszawski, and et al. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of Machine Learning and Systems*, volume 1, pages 27–39, 2019.

[40] Yihui He, Ji Lin, Zhijian Liu, and et al. AMC: automl for model compression and acceleration on mobile devices. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VII*, volume 11211 of *Lecture Notes in Computer Science*, pages 815–832. Springer, 2018.