Received: 24 March 2024 | Revised: 28 June 2024 | Accepted: 13 August 2024

DOI: 10.4218/etrij.2024-0139

SPECIAL ISSUE

ETRI Journal WILEY

NEST-C: A deep learning compiler framework for heterogeneous computing systems with artificial intelligence accelerators

Jeman Park[1] | **Misun Yu**[1] | **Jinse Kwon**[1] | **Junmo Park**[2] | **Jemin Lee**[1] | **Yongin Kwon**[1]

[1]Artificial Intelligence Computing Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

[2]Samsung Electronics, Hwaseong, Republic of Korea

Correspondence
Jemin Lee and Yongin Kwon, Artificial Intelligence Computing Research Laboratory, Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea.
Email: leejaymin@etri.re.kr and yongin.kwon@etri.re.kr

Funding information
This study is supported by a grant from the Institute of Information & Communications Technology Planning & Evaluation (IITP), funded by the Korean government (MSIT) (No. RS-2023-00277060, Development of OpenEdge AI SoC hardware and software platform).

Abstract

Deep learning (DL) has significantly advanced artificial intelligence (AI); however, frameworks such as PyTorch, ONNX, and TensorFlow are optimized for general-purpose GPUs, leading to inefficiencies on specialized accelerators such as neural processing units (NPUs) and processing-in-memory (PIM) devices. These accelerators are designed to optimize both throughput and energy efficiency but they require more tailored optimizations. To address these limitations, we propose the NEST compiler (NEST-C), a novel DL framework that improves the deployment and performance of models across various AI accelerators. NEST-C leverages profiling-based quantization, dynamic graph partitioning, and multi-level intermediate representation (IR) integration for efficient execution on diverse hardware platforms. Our results show that NEST-C significantly enhances computational efficiency and adaptability across various AI accelerators, achieving higher throughput, lower latency, improved resource utilization, and greater model portability. These benefits contribute to more efficient DL model deployment in modern AI applications.

KEYWORDS
AI accelerator, deep learning compiler, heterogeneous computing, model quantization, multi-level IR

1 | INTRODUCTION

In recent years, deep learning (DL) has revolutionized the field of artificial intelligence (AI), significantly impacting areas such as image recognition, natural language processing, and autonomous systems. DL frameworks such as PyTorch [1], ONNX [2], and TensorFlow [3] have significantly advanced the field by enabling the development and deployment of sophisticated neural networks.

However, these frameworks are primarily designed for general-purpose GPUs, which can lead to inefficiencies in specialized tasks. To address these inefficiencies, AI accelerators have been developed to maximize both throughput and energy efficiency compared to GPUs. Although AI accelerators are tailored for DL tasks, they

still rely on DL frameworks, creating a gap between these DL frameworks and AI accelerators can hinder the development and deployment of neural networks in fields with various constraints.

Traditional DL frameworks provide abstract functionalities but often lack the detailed optimizations required by various AI accelerators. To solve this problem, DL compilers have been developed [4]. These compilers take DL models developed within DL frameworks as inputs, optimize them, and translate them for specific hardware to produce efficient executable code. The optimization process includes graph optimization, memory management, parallel processing, quantization, and execution tuning. DL models are efficiently optimized to run on a diverse range of hardware, from datacenter servers to mobile devices and embedded IoT sensors. Leading DL compilers such as tensor virtual machine (TVM) [5], Glow [6], and XLA [7] offer specialized features and techniques for optimization and deployment across a wide range of applications. However, these existing solutions often fail to optimize the performance of newer AI accelerators such as neural processing units (NPUs) [8–12] and processing-in-memory (PIM) [13, 14] devices.

The rapid advancement of AI accelerators such as NPUs and PIM devices presents significant challenges. Traditional compilers, primarily designed for CPUs and GPUs, struggle to maximize the performance and leverage the unique features of these new accelerators. This gap necessitates a compiler framework that can efficiently optimize DL models for these diverse AI accelerators.

To overcome these challenges, this paper proposes the NEST compiler (NEST-C) [15], an advanced DL compiler framework designed to simplify deployment and enhance the efficient execution of DL models. As an open-source project, NEST-C generates optimized codes for various AI accelerators, including NPUs and PIM devices. Furthermore, NEST-C offers tuning features and tools tailored to the characteristics of each AI accelerator. The main contributions of NEST-C are as follows:

- To enable DL models for heterogeneous AI accelerators: NEST-C facilitates the use of DL models on a variety of AI accelerators, such as NPUs and PIMs, by supporting necessary adaptations for quantization and optimization. This ensures that models can efficiently operate across heterogeneous hardware platforms, broadening their applicability and performance.
- To optimize the DL model execution for multiple AI accelerators through graph partitioning: NEST-C utilizes graph partitioning techniques to efficiently distribute the workload of DL models. This is achieved by developing algorithms that enable dynamic task partitioning in mixed hardware systems, which include multiple AI accelerators, thereby maximizing computational efficiency.
- To enhance AI accelerator portability: By providing integration interfaces at each stage of the intermediate representation (IR) in the compilation process, NEST-C makes it easier to connect various hardware with compilers. This approach ensures that the compiler can be utilized more conveniently, facilitating seamless operation across different AI accelerators and enhancing the system's overall flexibility and efficiency.

2 | BACKGROUND AND RELATED WORK

2.1 | Common structures of DL compilers

The general design architecture of a DL compiler is divided into front-end and back-end architectures to facilitate optimization and support for various hardware platforms [4]. Additionally, there is an abstract representation known as IR. IR is a crucial concept in compiler design that acts as a bridge between high-level programming languages and machine code. This simplifies the complex process of translating human-readable code into instructions that can be executed by hardware. LLVM IR [16] is among the most commonly used IRs for targeting CPUs and GPUs, owing to its versatility and extensive support within the LLVM compiler. However, because it is a low-level representation, LLVM IR is not inherently suited for optimization at the level of DL operators, which often requires more abstract and higher level transformations to achieve efficient execution on various hardware platforms. Consequently, DL compilers require higher level IRs that are better suited for expressing and optimizing DL models. To accommodate the complexity and diversity of DL models and hardware platforms, DL compilers may employ multiple levels of IR, each serving a different stage of the compilation process. The front end handles DL models from frameworks such as TensorFlow and PyTorch, performing optimizations such as operation fusion, the elimination of redundant operations, and memory access optimization to improve efficiency. It then abstracts the model's structure and operations into standardized high-level IRs. In the front end, a high-level IR is used to optimize the relationships between operators and tensors in a hardware-independent manner. For example, TVM's

relay IR [17] uses tensors and placeholders for data representation, thereby providing scalability for various operators. The back end uses low-level IRs, which contain more hardware specifications than the higher layers. Based on this information, it optimizes the code to account for hardware-specific characteristics, calling hardware-specific libraries where necessary to maximize execution efficiency. In addition, the back end translates the code into code that can run on actual hardware (e.g., CPUs, GPUs, NPUs, and PIMs). In the back end, a low-level IR is utilized for hardware-dependent optimization and code generation for the target hardware. For instance, TVM advances the approach based on Halide, and Glow optimizes tensor processing with command-based IR.

2.2 | Existing DL compilers

Google's XLA enhances TensorFlow by providing hardware-agnostic and hardware-specific optimization through its compiler framework. It improves execution speeds and memory usage in DL models using techniques such as operator fusion and buffer analysis. XLA offers both just-in-time (JIT) and ahead-of-time (AOT) compilations, supporting diverse types of hardware such as CPUs and NVIDIA GPUs [18]. TVM, an open-source project, employs a multi-layered optimization strategy that separates computation from scheduling. It optimizes code across CPUs, GPUs, FPGAs, and ASICs using its unique IR systems RelayIR and tensor IR. Meta's Glow focuses on optimizing deep neural network models across various platforms using a two-stage IR process [19], prioritizing efficient execution and model portability. It supports CPUs and GPUs by optimizing memory usage and execution speed.

Accelerated Linear Algebra (XLA) is developed by Google for TensorFlow. It uses a compiler framework that performs hardware-agnostic high-level and hardware-specific low-level optimizations. The high-level optimizer (HLO) IR is used for graph-level optimizations, and LLVM is used for code generation. XLA optimizes TensorFlow graphs using techniques such as operator fusion, common subexpression elimination, and buffer analysis, resulting in improved execution speeds and memory usage for deep neural networks (DNNs). XLA optimizes TensorFlow graphs using techniques such as operator fusion, common subexpression elimination, and buffer analysis, resulting in improved execution speeds and memory usage for DNNs.

TVM is an open-source platform that introduces a multi-layered optimization approach. This approach separates computation from scheduling and is inspired by Halide. TVM's optimization process involves graph optimization, operator-level optimization, and automatic tuning, facilitated by a machine learning-based system to identify the optimal schedule among billions of possibilities. This approach enables efficient code generation for various targets including CPUs, GPUs, FPGAs, and ASICs. TVM's IR system, consisting of Relay and TIR, abstracts model operations and structures at different levels. This enables precise optimizations and code generation tailored to the specific characteristics of the hardware.

Graph Lowering (Glow) , developed by Facebook, aims to optimize DNN models for efficient execution across different hardware platforms using a two-stage IR process. The optimization involves high-level graph optimizations followed by hardware-specific optimizations through node lowering, focusing on minimizing memory consumption and maximizing execution speed. Glow's IR enables high-level graph optimizations and decomposes high-level operators into lower level linear algebra nodes, enabling efficient execution on CPUs, GPUs, and ASICs. This process prioritizes model portability while optimizing performance and uses an "in-memory form" lower level IR for hardware-dependent optimizations and memory latency hiding.

All existing compilers were primarily designed for CPUs and GPUs, making it challenging to adapt them for the newly emerging NPUs and PIMs. The arrival of complex and varied AI accelerators such as NPUs and PIMs has significantly increased the difficulty of optimizing edge devices that incorporate these accelerators. Moreover, they do not account for the complexities involved in the simultaneous optimization and leveraging of the parallel capabilities of various AI accelerators. Distinct differences exist between the current compilers (TVM, Glow, and XLA) and NEST-C, as detailed in Table 1. NEST-C includes features in each optimization category that are not supported by the existing compilers and, notably, provides broader support a wider range of NPUs.

3 | NEST-C

3.1 | NEST-C overview

This study devised NEST-C, a DL compiler designed to support various edge-specific AI accelerators, and provides optimizations including graph partitioning, quantization, and execution tuning. NEST-C supports traditional processing units such as CPUs and GPUs while generating optimized code for AI accelerators such

TABLE 1 Comparison of optimization features.

Optimization feature	TVM	Glow	XLA	NEST-C
Quantization	Calibration-based Int4, Int8, and Fp16	Profiling-based Int8, Int16, and Fp16	Profiling-based Int8, Int16, and Fp16	Profiling-based Int8, Int16, and Fp16; layer-wise mixed precision
Graph partitioning	Memory-based static partitioning	Static partitioning	Static and dynamic partitioning	Profiling-based dynamic partitioning
NPU back-end support	VTA (open architecture)	Habana (closed architecture)	Google TPU	EVTA, AimFuture NMP, OpenEdge Enlight, and SK Hynix GDDR6-AiM
Auto-tune of tile size	Execution time profiling, uniform tile size, and static tile scheduling	Execution time profiling	Auto-tuning and hardware-specific optimization	Hardware utilization, profiling based, ununiform tile size, and dynamic tile scheduling
Layer fusing and execution	Static layer fusing and layer-by-layer execution	Static layer fusing and layer-by-layer execution	Operator fusion for optimized execution	Dynamic layer fusing and dynamic layer execution
NN deployment on heterogeneous accelerators on a device	×	×	×	O
NN deployment on multiple devices	×	×	O	O

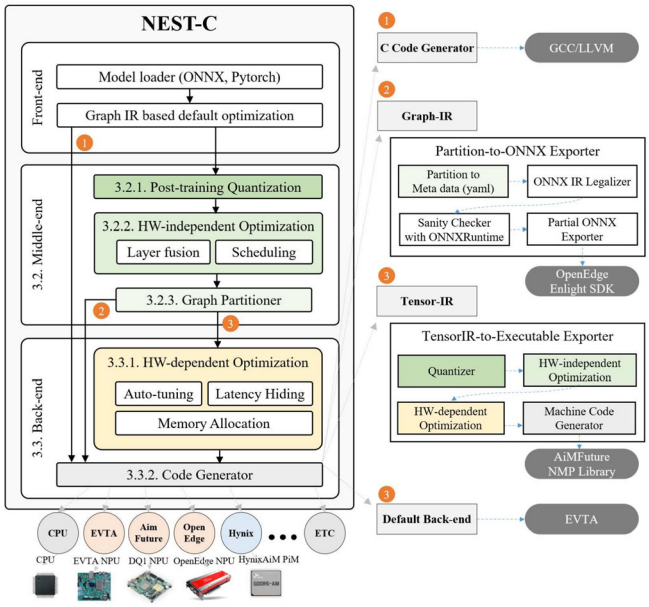

FIGURE 1 Architecture of the NEST-C ecosystem.

as NPUs and PIMs. Its architecture is divided into three main components: front end, middle end, and back end, as shown in Figure 1.

Initially, the front end converts the input from a DL model, defined within a framework such as ONNX or PyTorch, into a graph IR. NEST-C front end, which is fundamentally based on Glow's IR structure, executes basic optimizations (e.g., dead code elimination and transpose-node optimizations), which are also applied in Glow's graph IR.

Secondly, NEST-C uniquely supports the characteristics of resource-constrained edge AI accelerators through its middle end, distinguishing it from traditional DL compilers. The middle end performs hardware-independent optimization and graph partitioning. After receiving the graph IR from the front end, it subjects this IR to post-training quantization, layer fusion, and operator scheduling. These processes, which are adaptable to various AI accelerators, are executed before forwarding an IR graph to the graph partitioner. Graph partitioning then divides the computational graph of the model into subgraphs, each allocated to the processing unit that best matches its computational capacity, memory, and data-transfer speed requirements. This strategy optimally distributes model layers across hardware, enhancing execution efficiency, and facilitating parallel processing in edge environments.

Finally, in the back end, various hardware-dependent optimizations such as execution tuning, latency hiding, and memory allocation are applied to the partitioned subgraphs considering the characteristics of various edge AI

accelerators. NEST-C employs a tensor IR to perform hardware-dependent optimizations for each operator separately. This approach allows the fine-tuning of the performance of DL models on AI accelerators by optimizing memory usage, computational efficiency, and execution flow based on the unique characteristics and capabilities of each target device. The code generator creates executable code by utilizing a variety of back-end code generators, including the "C code generator," (CCodeGen) "LLVM IR generator," "relay IR generator," and "NPU code generator." Each code generator produces code optimized for specific processing units, which are eventually converted into executable files using compilers such as GCC, LLVM, TVM relay, and those provided by each back end.

3.2 | NEST-C middle end

3.2.1 | Post-training quantization and optimal configuration procedures in NEST-C

Quantization converts a neural network's FP32 precision to INT8, thereby reducing its memory footprint and accelerating inference. This technique maps floating-point values to a narrower integer range while maintaining the performance with minimal loss in accuracy. This enables the models to run efficiently on devices with limited computational resources.

In NEST-C, as shown in Algorithm 1, to achieve optimal quantization in terms of accuracy and latency, a variety of configurations are supported at the compiler level. Because quantization must be supported at the compiler level by default, the post-training quantization (PTQ) approach is followed. Similar to conventional PTQ methods, NEST-C involves a profiling stage for calibration before quantizing activations. Subsequently, based on the accuracy and latency requirements, configurations can be selectively created for the clipping, granularity, and mixed-precision schemes. Algorithm 1 is implemented at the compiler level. Therefore, it takes F as input, which is the graph IR of the DL model generated during the compilation phase. The output of Algorithm 1 is an IR F^* that includes the quantization information.

Quantization can reduce accuracy. To address this issue, the proposed algorithm sequentially applies schemes such as clipping, granularity, and mixed precision. If none of the configurations maintains the desired accuracy, it is considered that PTQ methods alone cannot solve this problem, and an error is output. In such cases, retraining the quantized model using external retraining tools is necessary.

Detailed explanations of each configuration selected by Algorithm 1 are provided below.

Algorithm 1 Procedure for quantization configuration

Input: Graph IR F
Output: Quantized Graph IR F^* or Error

Calibrate using sample images
Choose a scheme from (1), (2), (3), or (4)
if the accuracy is satisfactory **then**
    return F^*
end if
Select a clipping metric, either MAX or KL-Div.
if the accuracy is satisfactory **then**
    return F^*
end if
Choose the granularity, either layer-wise or channel-wise.
if the accuracy is satisfactory **then**
    return F^*
end if
Maintain precision in the first and last layers.
if the accuracy is satisfactory **then**
    return F^*
end if
return the error message: "Retrain the model with out-of-the-box tools."

Scheme: To enhance code generation efficiency across various hardware platforms, we emphasize uniform integer quantization, which incorporates four linear mapping techniques: asymmetric, symmetric, symmetric with UINT8, and symmetric power2. The notations used in these equations are as follows. Q_{i8} represents the quantized 8-bit integer value. V_{fp32} denotes the floating-point 32-bit value. S is the scale factor. Z indicates the zero point. V_{max} and V_{min} denote the maximum and minimum values, respectively. N symbolizes the bit width. Asymmetric (affine mapping) converts the float ranges to $[-2^{n-1}, -2^{n-1}-1]$, optimally utilizing the INT8 capacity. The quantization and dequantization processes are defined by

$$Q_{i8} = \mathrm{ROUND}\left(\frac{V_{f32}}{S} + Z\right), S = \frac{V_{\max} - V_{\min}}{2^n - 1}, \\ Z = -\mathrm{ROUND}\left(\frac{V_{\min}}{S}\right) - 2^{n-1}, V_{f32} = (Q_{i8} - Z)\cdot S. \quad (1)$$

Symmetric maps real zeros to quantized zeros without converting the FP32 range min and max, using the absolute maximum for setting qmin and qmax. Its quantization and dequantization are

$$Q_{i8} = \mathrm{ROUND}\left(\frac{V_{f32}}{S}\right), S = \frac{\mathrm{MAX}(\mathrm{ABS}(V_{f32}))}{2^{(n-1)} - 1}, \\ V_{f32} = S \cdot Q_{i8}. \quad (2)$$

The symmetric with UINT8 scheme blends asymmetric and symmetric methods, adapting to real-

value distributions and toggling between symmetric ($Z = 0$) and asymmetric ($Z = 128$) quantization as follows:

$$\begin{aligned} &Q_{i8} = \mathrm{ROUND}\left(\frac{V_{f32}}{S} + Z\right),\ S = \frac{\mathrm{MAX}(\mathrm{ABS}(V_{\mathrm{f32}}))}{2^n - 1}, \\ &Z = \{-128,\ \text{if } V_{\min} \geq 0,\ \text{otherwise}, \\ &V_{f32} = (Q_{i8} - Z) \cdot S \end{aligned} \quad (3)$$

The symmetric with a power of two simplifies the hardware design using bit-shift operations instead of multiplications and is defined by

$$S = 2^{\lceil \log_2 \frac{\mathrm{MAX}(\mathrm{ABS}(V_f))}{2^{(n-1)}-1} \rceil}. \quad (4)$$

Clipping: To mitigate accuracy loss without retraining, the Glow compiler employs clipping to minimize the Kullback–Leibler divergence between the floating-point and quantized distributions, addressing the impact of outliers in weight and activation distributions.

Granularity: The choice between tensor-wise and channel-wise quantization granularity balances accuracy and latency, with fine granularity increasing computational demand, particularly in convolutions with diverse weight values.

Mixed precision: NEST-C supports mixed-precision quantization by maintaining the first and last layers at their original precision (FP32) for execution. The first and last layers are known to be the most sensitive to quantization [20] and are prioritized for quick mixed-precision decisions. For additional mixed-precision applications, developers must determine the layers that should be quantized.

3.2.2 | Hardware-independent optimizations

Hardware-agnostic optimization focuses on the structural and algorithmic characteristics of DL models. This approach performs optimizations that can generally enhance performance across all types of hardware. In addition to the PTQ techniques previously mentioned (Section 3.2.1), the key hardware-agnostic optimization techniques applied in NEST-C include layer fusion and operator scheduling.

Layer fusion: Fusion, such as combining convolution with batch normalization and ReLU activation, serves to increase computational efficiency and minimize memory usage. The convolution operation, denoted by $y = w * x + b$, when fused with batch normalization, transforms into $y' = \gamma[(w * x + b - \mu)/\ (\sqrt{\sigma^2 + \epsilon})] + \beta$, streamlining the computation by merging the standardization directly into the convolution process. Similarly, fusing convolution with ReLU activation simplifies the operation to $y' = \max(0,\ w * x + b)$ by applying nonlinearity immediately after convolution, thereby reducing computational steps and memory overhead. Layer fusion, implemented as an optimization pass at the compiler level, streamlines neural network operations by reducing computational steps and memory usage, enhancing efficiency and speed, especially in memory-restricted environments.

Operator scheduling: From the perspective of IR scheduling in neural network computation, the scheduling of operators is crucial for optimizing performance. Transformations, such as in-place buffer modifications for element-wise arithmetic, are key examples of such optimizations. In this context, instructions manipulate global variables or locally allocated buffers, with each operand annotated using the qualifiers `in`, `out`, or `inout`. These qualifiers indicate whether a buffer is read from (`in`), written to (`out`), or both (`inout`), indicating to the optimizer when optimizations such as copy elimination or buffer sharing, are possible.

3.2.3 | Graph partitioner (PartitionTuner)

Typically, edge AI accelerators support a limited range of DL operations, necessitating the partitioned execution of DL models. Frameworks such as XLA and TVM provide code generation for accelerators but are limited to supporting only a single accelerator. To address this issue, NEST-C integrates PartitionTuner [21], which distributes the operations of a DL model across multiple accelerators and synchronizes the processing of their operations to enhance the overall performance. Figure 2 illustrates the architecture of this approach.

Initially, the Branch Extractor analyzes the structure of the input graph to identify branches that require sequential execution. Subsequently, the Graph Partitioner segments the graph into groups of operations for each branch. The Profiler then generates a machine

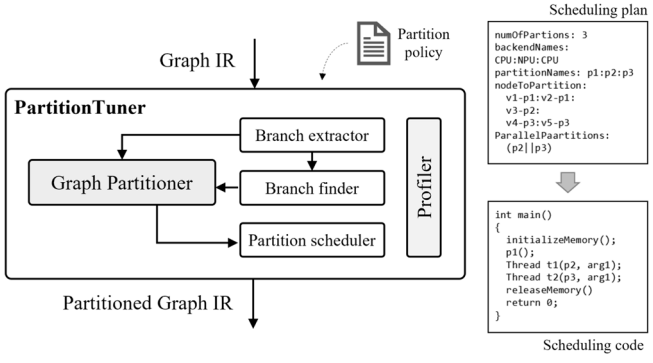

FIGURE 2 Workflow of the PartitionTuner in NEST-C.

code for each back end identified by the Back-End Finder and profiles the execution times for each group of operations. Based on these profiling results, a user-modifiable Partition Policy file is automatically generated. The Partition Scheduler creates partitions based on the Partition Policy using the back-end information assigned to the operations. It then develops a Scheduling Plan that specifies the execution sequence among the partitions. Subsequently, based on the Scheduling Plan, the Partition Tuner generates Scheduling Code for each back end. This approach accurately identifies the fastest optimal partition for each accelerator, ensuring the optimal utilization of various accelerators within heterogeneous computing systems. Additionally, the overall execution speed can be significantly improved by employing independently operable accelerators in parallel.

3.3 | NEST-C back end

3.3.1 | Tensor IR and hardware-dependent optimization

During graph partitioning, the hardware on which the partitioned graph will execute is determined, and based on this specific hardware, the IR is lowered to the tensor-IR level. At this level, the memory size and location of the inputs and outputs used by each operator are determined. This phase aims to find the optimal execution code that minimizes memory usage while improving execution performance. NEST-C can dynamically allocate the main memory with the help of the operating system depending on the target hardware. However, this method introduces overhead for allocation and deallocation. Furthermore, if the DL tensor data are distributed across various memory areas, it could degrade the performance of the NPU's direct memory access.

Therefore, NEST-C calculates the total memory size required to execute the operations of a specific partition from beginning to end. Subsequently, it allocates that amount of memory—preferably a contiguous block—just once. It then devises a strategy to statically reuse memory within that allocated space without further assistance from the operating system. This approach reduces memory fragmentation and allocation overhead, ensuring efficient memory management and optimized performance for the DL computations on the targeted hardware.

NPUs have small internal buffers so that they can deliver data faster than DRAM; however, they cannot store all inputs and outputs simultaneously. Instead, inputs from memory are partially stored, with outputs accumulated as the computations proceed. This necessitates diverse scheduling strategies such as input stationary, weight stationary, and output stationary [22]. "Tiling" involves storing and processing inputs and outputs in the buffer sequentially; an NPU requires all inputs for a tile to be loaded and space for outputs to be available before processing begins.

The period an NPU waits to begin operation is called the "memory latency," and strategies such as adjusting the size of tiles and employing double buffering are used to reduce this time, a practice known as "memory latency hiding." NEST-C provides three main optimization approaches to maximize memory latency hiding and determine the optimal tile size and scheduling.

Empirical discovery through device profiling: This involves experimenting on actual devices to find optimal settings. However, an exhaustive search of all the possible configurations can be time-consuming or inefficient.

Auto-tuning using machine learning: To mitigate the challenges of exhaustive search, NEST-C employs auto-tuning techniques that leverage machine learning to streamline the optimization process, which significantly reduces the search space.

Utilization of performance modeling or performance accurate simulators: This method involves using performance models or simulations to search all cases. If a model or simulator is implemented precisely, the optimal execution method can be identified quickly.

3.3.2 | Code generator

The code generator creates code tailored to the target hardware based on the optimized tensor IR. Depending on the target hardware, it may need to generate binary machine code or code that conforms to the device driver or inference library interface of the target hardware. Alternatively, the tensor IR could be lowered to an LLVM IR via an LLVM code generator.

It is necessary to implement a unique code generator for each target hardware, and NEST-C provides references that can be consulted when developing code generators for new hardware. Additionally, NEST-C offers functionality that not only generates code for several popular models (e.g., ImageNet classification) but also automatically creates code for input preprocessing and output presentation. The code generator further includes various back-end generators, such as C, NPU, and EVTA code generators. The following section details the integration of the code generator with diverse AI accelerators across each multi-level IR.

4 | DL COMPILER IMPLEMENTATION AND OPTIMIZATION FOR DIVERSE TARGET HARDWARE

NEST-C enables the export of DL models to an LLVM IR, thereby supporting a wide range of general-purpose processors that LLVM supports, including prominent CPU architectures from Intel and ARM. However, NEST-C primarily targets accelerators (NPUs) specifically developed for DL computations, and the compiler implementation can be categorized into three forms based on the hardware characteristics and interface levels provided by these NPUs, as illustrated in Figure 1.

Using only the front end (1): This approach involves generating code directly from the initially created graph IR without any optimization process such as C code generation (Section 4.1).

Utilizing the middle end (2): This involves performing optimizations such as PTQ and layer fusion, generating a partitioned graph, and then using the target hardware's development toolkit for the remaining compilation process such as OpenEdge Enlight [9] (Section 4.2).

Employing the full stack of NEST-C (3): This approach generates executable final code for the target hardware directly from the optimized tensor IR that has undergone all of NEST-C's processes and optimizations (e.g., AimFuture's NMP [8]).

The development and verification of the full NEST-C stack were initially performed using EVTA (Section 4.4), which served as a reference NPU. AiMFuture NMP (Section 4.3) undergoes a compilation process similar to that of EVTA. OpenEdge Enlight (Section 4.2) implements the compiler up to the middle end of NEST-C. Additionally, implementations utilizing NEST-C include SK Hynix's GDDR6-AiM [13] and TVM's Relay, showcasing the flexibility and adaptability of NEST-C to a broad spectrum of computational platforms for DL.

4.1 | CCodeGen: C code generation

The CCodeGen automatically generates C/C++ code compatible with common cross-compilers, such as GCC and LLVM directly from the output of the DL framework. This is important when deploying DL models on heterogeneous computing systems. In heterogeneous computing systems with varying requirements, developers must manually compile, translate, and deploy DL models into machine code or back-end inputs for each hardware type. This process places a heavy burden on developers. Therefore, CCodeGen generates C/C++ code that can be used by most computing systems, making it easier for developers to support heterogeneous devices. CCodeGen parses the input DL model to generate a dataflow graph (DFG), which encompasses the operations, input/output information, and parameters of the DL model. Next, using the DFG, it generates "inference.c" and "inference.h," C code files that contain inference functions, and "weight.bin," a binary file that contains the trained weights used for inference. Figure 3 illustrates an example of the operation library. The API is defined using C++ template functions to ensure that the operation functions are not dependent on the variable types (such as float, INT16, and INT8). Additionally, it allows users to select the type of compiler provided by the device (GCC or LLVM), automatically handling any differences in syntax between compilers. Moreover, it includes the option to decide on the use of different libraries, such as OpenBLAS.

4.2 | Graph-IR implementation

Chip vendors provide their own back-end compilers for commercial-grade AI semiconductors. Therefore, to reduce the development effort and generate high-quality final target code, it is necessary to integrate NEST-C with compilers provided by chip vendors. NEST-C offers ONNX conversion functionality for integration with private back-end compilers at the graph-IR level. As explained previously, hardware-independent optimizations are performed at graph-IR level. Then, the graph IR is serialized and stored according to the YAML meta-format. After normalizing the stored meta file to align with the ONNX format, an ONNX suitable for input to the private compiler provided by commercial-grade AI semiconductor manufacturers is generated. As shown in Figure 1, for the conversion from graph IR to ONNX, NEST-C includes four functional modules: Partition to YAML, ONNX IR Legalizer, Sanity Checker with ONNX-Runtime, and Partial ONNX Exporter.

Partition to YAML: This module divides the input model into partitions of the desired size. A partition can

op_inference.h

```
//GCC define
#define USE_GCC
//OpenBLAS define
#define USE_OpenBLAS
//using MAC
#ifdef __MACH__
..

void convolution(..)
void matmul(..)
void avgpool(..)
void maxpool(..)
void add()
..
```

op_inference.c

```
void convolution(..) {
#ifdef USE_GCC /*GCC*/
..
#else  /* LLVM */
..
#endif
..
/*for matmul*/
#ifdef OpenBLAS
/* with OpenBLAS */
..
#else
/* without OpenBLAS */
..
```

FIGURE 3 API and code example of the operation library.

represent either an entire model or a specific group of consecutive operators. The generated YAML file includes information regarding each partition and the names of the operators used in the IR graph. This information is utilized for code generation tailored to the partitioning or integration with specific back-end compilers.

ONNX IR Legalizer: This module performs normalization to resolve representation differences between the graph IR and the ONNX IR. This step is necessary for converting the graph IR into ONNX IR while ensuring consistency in the representation.

Sanity Checker with ONNXRuntime: To verify that the model converted to ONNX functions correctly, this module uses ONNXRuntime, a compiler for general-purpose CPUs, to validate the model's results.

Partial ONNX Exporter: This module converts and stores the YAML meta file in the ONNX format. This step converts the metafile, after hardware-independent optimization, to the final ONNX format.

Using this structure, NEST-C partitions and normalizes the input model before generating the final ONNX format model. This model undergoes hardware-independent optimization and is integrated with the Enlight compiler, enabling hardware-accelerated inference. This approach also allows for the integration of a third-party private compiler.

4.3 | Tensor-IR implementation

The neuromorphic processor (NMP) is an embedded evaluation board developed by LG Electronics centered on a novel architecture designed to facilitate efficient DL operations. The foundational principle of the NMP's design involves leveraging RISC-V instruction set architecture extensions to create specialized instructions for various CNN components, including convolutional layers, fully connected layers, pooling layers, and element-wise operations. The architecture of the NMP features a multicore NPU comprising several processors. Each processor houses multiple processing units, each equipped with a RISC-V core, a multiply-accumulate unit, and memory buffers to support the computational demands of neural network processing.

Quantization for NMP NMP is a hardware accelerator that supports only integers in Q-format. Therefore, NEST-C middle end for NMP must ensure that all DL models are quantized into Q-format. For Q-format quantization, the minimum and maximum values of all tensors holding floating-point values are calculated, and NEST-C determines the most suitable format, either INT8 or INT16, as supported by NMP. This process allows the precise representation of floating-point numbers within the fixed-point integer constraints of the NMP, thereby optimizing both the accuracy and efficiency of DL computations.

Back-end optimization for NMP Because each processing unit in NMP possesses its own buffer and accelerator, operations must be well tiled and distributed, similar to EVTA. Whereas EVTA determines the optimal tile size and number of buffers by profiling the execution performance on real devices considering only output stationary scheduling, NMP considers input stationary, weight stationary, and output stationary scheduling. It uses a performance model to exhaustively search for and identify the best performance across all scenarios. This comprehensive approach enables NMP to optimize the resource allocation and processing efficiency for diverse computational patterns in DL operations.

4.4 | EVTA

EVTA is a custom NPU based on the VTA [23] from the Apache TVM project. It modifies VTA's compute module to include an MOV instruction, reducing power usage and improving performance by eliminating DRAM data transfers between the output and input buffers. Moreover, EVTA supports diverse operations (INT8, FP16, FP32, and binary) and allows multiple NPUs to work together and share DRAM resources, as illustrated in Figure 4.

EVTA's IR is extended within the structure of NEST-C's graph IR and tensor IR to meet the hardware characteristics and optimization requirements.

Graph-IR and middle-end optimization At the middle end of NEST-C, efforts are made to minimize DRAM access and maximize hardware utilization by changing the data layout according to the DL operators and performing layer fusion optimizations. The compute module of EVTA can process ReLU operations in conjunction with matrix multiplication; hence, ReLU is fused with convolution and fully connected operators. Because EVTA can

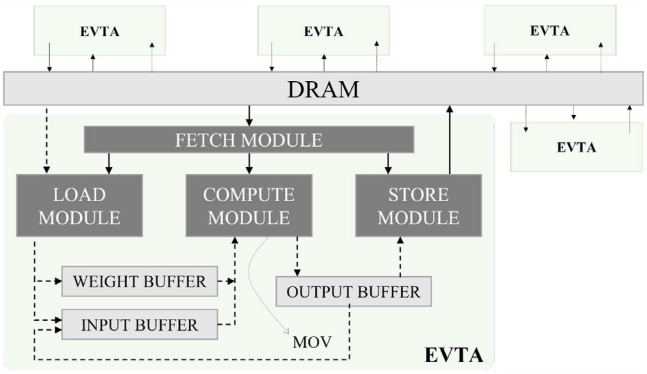

FIGURE 4 EVTA architecture and configuration.

only perform matrix multiplication during the middle-end stage, it transforms a 4-dimensional input into a 6-dimensional one, converting the last two dimensions into a matrix size that EVTA can process in one cycle. Additionally, EVTA's graph IR was implemented to represent such a six-dimensional data layout. Following this, graph partitioning is performed according to the configuration of multi-EVTA, and it is then lowered to tensor IR.

Tensor IR and back-end optimization In NEST-C back end, as the graph IR is lowered to tensor IR, the execution order of each operator and the DRAM storage locations for the operator's input and output data are determined. For EVTA, the inputs must be tiled because the buffer size is limited. Double buffering is utilized to maximize memory latency hiding. The extent of latency hiding varies with the tile size and number of buffers, and NEST-C employs profiling and auto-tuning to determine the optimal tile size and number of buffers.

Codegen NEST-C provides the EVTA execution library along with the device driver, featuring an interface in the format of Figure 5. NEST-C generates the code according to the execution library interface. For convolution operations, the arguments also include optimized tile sizes and the number of buffers. The generated code is designed to allow the device driver to parse the arguments at runtime and immediately execute the code.

NEST-C is tasked with handling a large number of tensor data and deep layers. The code generated by NEST-C computes each layer, and the results are temporarily stored in DRAM. To ensure that the EVTA developed using NEST-C is error free, it is necessary to validate the computational results with respect to the expected values. For this purpose, NEST-C supports a debugging mode that allows the temporary results stored in DRAM for each layer to be outputted to a file. The results of the operations may differ slightly, depending on the hardware and data type. NEST-C accommodates these differences by allowing the tolerance level to be set.

5 | EXPERIMENTS AND EVALUATION

The performance of AI accelerators and heterogeneous systems using NEST-C was evaluated in three experiments. First, we conducted an EVTA with PartitionTuner experiment to demonstrate the reduction in latency and improved computational efficiency when using both CPU and NPU resources with NEST-C. Next, to verify the portability of NEST-C, we tested it with two types of commercial AI accelerators: AimFuture NMP for the tensor-IR interface and OpenEdge Enlight for the graph-IR interface. These experiments encompassed a variety of DL models trained on ImageNet [24], including GoogLeNet [25], ResNeXt50 [26], ResNet50 [27], ResNet18 [27], MNIST [28], LeNet [29], MobileNetV2 [24], and SqueezeNet [30]. Considering the diversity in the types of operations supported by each AI accelerator, distinct DL models were chosen for each experiment.

```
int convolution( int8_t * input, int8_t *kernel,
    int32_t *bias, int8_t *output, ..., int
    nVirtualThread, int tileHSize, int tileWSize
    );
```

FIGURE 5 EVTA execution library interface for convolution.

5.1 | EVTA with PartitionTuner

The Xilinx ZCU102 platform was used to evaluate the performance of employing EVTA in NEST-C. The ZCU102 provides a quad-core ARM Cortex-A53 CPU and FPGA. Four EVTAs were ported to the FPGA on the Xilinx ZCU102 platform to run at 333 MHz. CCodeGen was used to generate the machine code for the CPU. The model was partitioned by CPU and NPU using PartitionTuner, which also performed the quantization for the EVTAs. Table 2 shows a significant reduction in latency when NPUs are used in conjunction with the CPU instead of using the CPU alone. However, increasing the number of NPUs does not significantly reduce latency. This is because quantization and data dimension transformation operations required to use NPUs are executed on the CPU, which is much slower than the NPUs. The parallel execution of these operations by two NPUs and the CPU limits fully parallel processing. Therefore, to improve the performance of multiple NPUs, it is important to partition and schedule the model.

TABLE 2 Latency (ms) of the DL models on multi-EVTA.

	Baseline	NEST-C (EVTA)	
	CPU	CPU + NPU	CPU + 2NPUs
Resnet18	1262.45	247.41	125.09
Resnet50	3180.52	654.33	580.38
ResNext50	6717.05	1645.90	756.45
GoogLeNet	1455.75	497.52	287.94
SqueezeNet	297.56	175.98	97.91
AlexNet	884.60	789.10	-
EfficientNet	6178.70	2820.72	-
ZFNet512	1673.95	1662.54	-
MNasNet	1258.35	866.04	-

5.2 | Tensor-IR interface evaluation

To assess the performance of AI accelerators at the tensor-IR level in NEST-C, the NMP [22] board provided by AimFuture was utilized. Various DL models were evaluated, including MNIST, LeNet, ResNet18, ResNet50, SqueezeNet, and Inception. Performance evaluations for each DL model within NEST-C were conducted and compared with the performance data from XLA provided by NMP. The model accuracy provided by Google TensorFlow was used as baseline. Table 3 shows that most of the reference models exhibited results similar to the baseline accuracy, suggesting that NEST-C was effectively implemented at the tensor-IR level within NMP. Unlike NMP's XLA, which is optimized directly by the hardware manufacturer, NEST-C automatically performs optimization at the compiler level through hardware profiling. Despite this, the latency performance of NEST-C was very close to that of NMP's XLA, indicating its ability to maintain optimal performance even when new AI accelerators were integrated.

5.3 | Graph-IR interface evaluation

The proposed common ONNX-based interface was used to experimentally validate the successful linkage between the general AI compiler and the private NPU compiler. For a comparative evaluation, the results generated by integrating Enlight and NEST-C were compared with the performance results of Resnet50 and MobileNetV2 DL models on general hardware CPUs and GPUs supported by the existing NEST-C.

OpenEdge Enlight supports various layer types, including convolution layers with kernel sizes ranging from 1×1 to 7×7 and strides from 1 to 4. The depth-wise convolution layer supports a 3×3 kernel with strides ranging from 1 to 2. The supported activation functions include Bypass, ReLU, Leaky ReLU, Sigmoid, Tanh, and Mish. Additionally, pooling operations are supported with configurations of 4:1 (2×2) and Stride 2, which are available in max/average and global average pooling modes.

The experimental results, as shown in Figure 6A, indicate a decrease in accuracy when models are executed on OpenEdge Enlight. This decrease was due to the quantization process, which reduced the model precision from 32 to 8 bits. Despite this reduction, the models maintained acceptable performance.

In terms of the inference latency, as depicted in Figure 6B, the integration of NEST-C and OpenEdge Enlight achieved impressive inference times of 9 ms for MobileNetV2 and 107 ms for Resnet50. This performance is significantly faster than the inference times on an Intel i7 CPU, which are 59.9 ms for MobileNetV2 and 136.9 ms for Resnet50. It is also comparable to the results on an NVIDIA 2080ti GPU, which are 15.35 ms for MobileNetV2 and 14.8 ms for Resnet50.

5.4 | Discussion and limitations

The experimental results demonstrated the effectiveness and versatility of the NEST-C framework across various

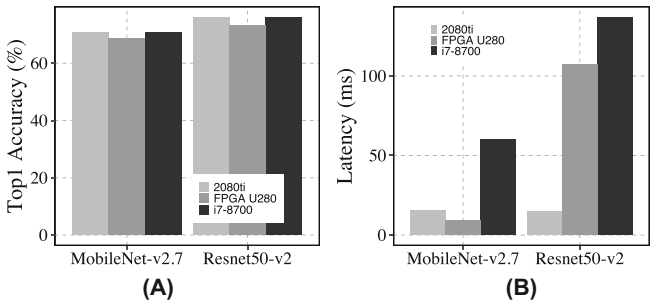

FIGURE 6 Accuracy and latency of the models on the targets (Intel i7-8700 CPU, 2080ti GPU, and U280 FPGA-based NPU).

TABLE 3 Accuracy and latency of the DL models on NMP.

	NEST-C (NMP)		XLA (NMP)		TensorFlow
	Accuracy Top 1	Latency (ms)	Accuracy Top 1	Latency (ms)	Accuracy Top 1
MNIST	98.2	0.163	-	-	98.90
LeNet	94.8	0.635	99.9	0.199	94.80
Resnet18	66.8	22.477	-	-	69.93
Resnet50	71.3	67.385	70.7	55.927	74.93
SqueezeNet	48.7	14.269	47.1	12.504	49.00
Mobilenet	70.2	70.077	70.9	14.03	71.80
Inception	74.2	30.59	76.9	72.686	77.90

AI accelerators, showing significant improvements in latency reduction and computational efficiency. The portability of NEST-C was validated through tests using AimFuture NMP and OpenEdge Enlight, which maintained optimal performance across different hardware platforms. However, owing to the constraints of using commercial hardware, we were limited to experimenting with DL models provided by hardware manufacturers, which restricted the range of models tested. Additionally, there is a potential accuracy loss owing to quantization from 32 to 8 bits, and further evaluation is needed to understand the scalability of NEST-C with large models and datasets. Addressing these limitations in future studies will enhance the robustness and efficiency of NEST-C for diverse AI applications.

6 | CONCLUSIONS

This study developed NEST-C, a novel DL compiler framework designed to improve the deployment and performance of DL models across various AI accelerators. The experimental results demonstrate that NEST-C significantly enhances computational efficiency and adaptability, achieving higher throughput and lower latency through profiling-based quantization, dynamic graph partitioning, and multi-level IR integration. Despite the limitations of testing with a limited range of DL models owing to commercial hardware constraints, NEST-C maintained its optimal performance across different hardware platforms. Despite being constrained to a limited range of DL models provided by hardware manufacturers, NEST-C demonstrated its potential for broader applicability.

Future work will focus on addressing the current limitations by extending the range of tested models, minimizing accuracy loss due to quantization, and evaluating the scalability of NEST-C with larger models and datasets. By overcoming these challenges, we aim to further enhance the robustness and efficiency of NEST-C, ensuring its adaptability to a wider array of hardware platforms and DL tasks. Additionally, NEST-C was primarily designed to optimize DNNs used for image recognition. Therefore, we plan to extend its capabilities to support large-language models and transformer architectures to suit the evolving DL research environments.

CONFLICT OF INTEREST STATEMENT

The authors declare that there are no conflicts of interest.

ORCID

Jeman Park https://orcid.org/0009-0002-9524-0738
Misun Yu https://orcid.org/0000-0001-7319-1053
Jinse Kwon https://orcid.org/0000-0003-3091-9926
Junmo Park https://orcid.org/0000-0002-8500-8874
Jemin Lee https://orcid.org/0000-0002-9332-3508
Yongin Kwon https://orcid.org/0000-0003-2973-246X

REFERENCES

1. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: an imperative style, high-performance deep learning library*, (Proc. 33rd Int. Conf. Neural Inf. Process. Syst., Vol. **32**, Curran Associates Inc., Red Hook, NY, USA), 2019.
2. ONNX Contributors, *Open Neural Network Exchange (ONNX)*, 2024. https://github.com/onnx/onnx,_2024. Accessed: 2024-03-18.
3. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard, *TensorFlow: a system for large-scale machine learning*, (12th USENIX Symp. Operating Syst. Des. Implementation (OSDI'16)., Savannah, GA, USA), 2016, pp. 265–283.
4. M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, *The deep learning compiler: a comprehensive survey*, IEEE Trans. Parallel Distrib. Syst. **32** (2020), no. 3, 708–727.
5. T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, and L. Ceze, *TVM: an automated end-to-end optimizing compiler for deep learning*, (13th USENIX Symp. Operating Syst. Des. Implementation (OSDI'18), Carlsbad, CA, USA), 2018, pp. 578–594.
6. N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, and R. Levenstein, *Glow: graph lowering compiler techniques for neural networks*, arXiv preprint, 2018. https://doi.org/10.48550/arXiv.1805.00907
7. C. Leary and T. Wang, *XLA: TensorFlow, compiled*, 2017. TensorFlow Dev Summit.
8. AiM Future, *The future of artificial intelligence: AiM future's product lineup*, 2023. https://aimfuture.ai. Accessed: 2024-03-22.
9. OPENEDGES, *Neural processing unit (NPU) IP—ENLIGHT*, 2022. URL https://www.openedges.com/npu. Accessed: 2024-03-22.
10. J.-W. Jang, S. Lee, D. Kim, H. Park, A. S. Ardestani, Y. Choi, C. Kim, Y. Kim, H. Yu, H. Abdel-Aziz, J.-S. Park, H. Lee, D. Lee, M. W. Kim, H. Jung, H. Nam, D. Lim, S. Lee, J.-H. Song, S. Kwon, J. Hassoun, S. Lim, and C. Choi, *Sparsity-aware and re-configurable NPU architecture for Samsung Flagship Mobile SoC*, (ACM/IEEE 48th Annu. Int. Symp. Comput. Archit., Valencia, Spain), 2021, pp. 15–28.
11. Qualcomm, *Unlocking on-device generative AI with an NPU and heterogeneous computing*, 2024. https://www.qualcomm.com. Accessed: 2024-03-22.
12. Apple, *Deploying transformers on the Apple Neural Engine*, 2023. https://machinelearning.apple.com/research/deploying-transformers-on-the-apple-neural-engine. Accessed: 2024-03-22.
13. Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, and B. An, *System architecture and*

software stack for GDDR6-AiM, (IEEE Hot Chips 34 Symp., Cupertino, CA, USA), 2022, pp. 1–25.
14. Samsung, *HBM-PIM: cutting-edge memory technology to accelerate next-generation AI*, 2023. https://semiconductor.samsung.com/. Accessed: 2024-03-18.
15. ETRI, NEST-C. https://gitlab.com/ones-ai/nest-compiler. Accessed: 2024-03-22.
16. C. Lattner and V. Adve, *LLVM: a compilation framework for lifelong program analysis & transformation*, (Int. Symp. Code Gener. Optim., San Jose, CA, USA), 2004, pp. 75–86.
17. J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, Relay: a new IR for machine learning frameworks, (Proc. 2nd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang., Association for Computing Machinery, Philadelphia, PA, USA), 2018, pp. 58–68.
18. J. Dean, *Machine learning for systems and systems for machine learning*, *Presentation at 2017 Conf. Neural Inf. Process. Syst.*, Curran Associates, Long Beach, CA, USA, 2017.
19. Meta, *Glow's Graph IR optimization*. https://github.com/pytorch/glow/blob/master/docs/Optimizations.md. Accessed: 2024-02-22.
20. J. Lee, M. Yu, Y. Kwon, and T. Kim, *Quantune: Post-training quantization of convolutional neural networks using extreme gradient boosting for fast deployment*, Future Gener. Comput. Syst. **132** (2022), 124–135.
21. Y. Misun, K. Yongin, L. Jemin, P. Jeman, P. Junmo, and K. Taeho, *PartitionTuner: an operator scheduler for deep-learning compilers supporting multiple heterogeneous processing units*, ETRI J. **45** (2023), no. 2, 318–328.
22. R. Sousa, M. Pereira, Y. Kwon, T. Kim, N. Jung, C. S. Kim, M. Frank, and G. Araujo, *Tensor slicing and optimization for multicore NPUs*, J. Parallel Distrib. Comput. **175** (2023), 66–79.
23. T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, *A hardware-software blueprint for flexible deep learning specialization*, IEEE Micro **39** (2019), no. 5, 8–16.
24. J. Deng, W. Dong, and R. Socher, *ImageNet: a large-scale hierarchical image database*, (IEEE Conf. Comput. Vision Pattern Recognit., Miami, FL, USA), 2009, pp. 248–255.
25. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, (Proc. IEEE Conf. Comput. Vision Pattern Recognit. (CVPR), Boston, MA, USA), 2015, pp. 1–9.
26. S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, *Aggregated residual transformations for deep neural networks*, (IEEE Conf. Comput. Vision Pattern Recognit. (CVPR), Honolulu, HI, USA), 2017, pp. 1492–1500.
27. K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, (IEEE Conf. Comput. Vision Pattern Recognit. (CVPR), Las Vegas, NV, USA), 2016, pp. 770–778.
28. L. Deng, *The MNIST database of handwritten digit images for machine learning research*, IEEE Signal Process. Mag. **29** (2012), no. 6, 141–142.
29. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proc. IEEE **86** (1998), no. 11, 2278–2324.
30. F. N. Iandola, S. Han, and M. W. Moskewicz, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*, arXiv preprint, 2016. https://doi.org/10.48550/arXiv.1602.07360

AUTHOR BIOGRAPHIES

Jeman Park received his BS, MS, and PhD degrees in Electronics and Computer Engineering from Hanyang University, Republic of Korea, in 2004, 2006, and 2014, respectively. He is a senior researcher at the Electronics and Communications Research Institute, Daejeon, Republic of Korea. His research interests include computer networks, edge computing, and deep learning compilers.

Misun Yu received the MS degree from the Department of Computer Science and Engineering at Pohang University of Science and Technology, Republic of Korea. She is a principal researcher at the Electronics and Communications Research Institute Daejeon, Republic of Korea. Her main research interests include concurrent program analysis, software testing, deep learning, and embedded systems.

Jinse Kwon received MS and PhD degrees in Computer Science and Engineering from Chungnam National University, Daejeon, Republic of Korea in 2017 and 2024, respectively. He is currently a researcher at the Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea. His research interests include deep learning compilers and on-device computing.

Junmo Park received the BS degree in Computer Science from Kwangwoon University, Seoul, Republic of Korea in 2012 and the MS degree from the Graduate School of Convergence Science and Technology at Seoul National University, Republic of Korea, in 2020. He joined Samsung Electronics in Hwaseong, Republic of Korea, in 2012, where he has been involved in compiler optimization and development. Since 2020, he has been working as a Principal Software Engineer on mobile GPU compilers. His research interests include deep learning, compilers, embedded systems, HW/SW co-design, and optimization.

Jemin Lee received his BS and PhD degrees in Computer Science and Engineering from Chungnam National University, Daejeon, Republic of Korea, in 2011 and 2017, respectively. He is currently a senior researcher at the Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea. Since 2023, he has also served as an assistant professor in the AI Department at the University of Science and Technology, Daejeon, Republic of Korea. He was a postdoctoral researcher at the Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea from 2017 to 2018. His research interests include energy-aware mobile computing and deep learning compilers.

Yongin Kwon received the BSc degree in Electrical and Electronic Engineering from the Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea, in 2008, and MS and PhD degrees in Electrical and Computer Engineering from Seoul National University, Republic of Korea, in 2010 and 2015, respectively. From 2015 to 2019, he worked for Samsung Electronics as a Staff Software Engineer. Since 2019, he has been with the Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea, where he is currently a senior researcher. His research interests include neural processing units, compilers, deep learning, and embedded systems.

How to cite this article: J. Park, M. Yu, J. Kwon, J. Park, J. Lee, and Y. Kwon, *NEST-C: A deep learning compiler framework for heterogeneous computing systems with artificial intelligence accelerators*, ETRI Journal **46** (2024), 851–864, DOI 10.4218/etrij.2024-0139