Intelligent Computing
A SCIENCE PARTNER JOURNAL

REVIEW ARTICLE

Compiler Technologies in Deep Learning Co-Design: A Survey

Hongbin Zhang[1,2], Mingjie Xing[1], Yanjun Wu[1,3*], and Chen Zhao[1,3]

[1]Institute of Software, Chinese Academy of Sciences, Beijing, China. [2]University of Chinese Academy of Sciences, Beijing, China. [3]State Key Lab of Computer Science, Beijing, China.

[*]Address correspondence to: yanjun@iscas.ac.cn

Citation: Zhang H, Xing M, Wu Y, Zhao C. Compiler Technologies in Deep Learning Co-Design: A Survey. *Intell. Comput.* 2023;2:Article 0040. https://doi.org/10.34133/icomputing.0040

Submitted 29 March 2023
Accepted 29 May 2023
Published 19 June 2023

With the rapid development of deep learning applications, general-purpose processors no longer suffice for deep learning workloads because of the dying of Moore's Law. Thus, computer architecture innovation has entered a golden age for domain-specific design, which has led to a demand for new compilation technologies to facilitate cross-layer optimization. Historically, hardware and software have been collaboratively designed. Today, these co-design ideas still benefit the deep learning field in both academia and industry, encompassing additional aspects and layers. In this study, we elaborate on past and recent works on deep learning compilers and co-design while focusing on the combination of these two technologies, which we believe is the trend in the new deep learning era. After summarizing the existing compilation technologies and co-design approaches, we propose a domain-specific compilation framework, the Buddy Compiler, for a typical deep learning co-design system.

Introduction

With the downtrend observed in Moore's law, general-purpose processors cannot meet the performance and power requirements of deep learning workloads. Consequently, both industry and academia are engaged in hardware–software co-design, and compiler technology is one of the key components. Historically, deep learning software and hardware have promoted each other. Advances in hardware have allowed deep learning algorithm upgrades, and the characteristics of these algorithms have inspired hardware innovation. Deep learning models are growing in both size and complexity. The number of model parameters has increased from 60 million to 175 billion from AlexNet [1] to ChatGPT [2], with the applications expanding from image and speech recognition to artificial intelligence of things (AIoT), robotics, recommendation systems, and multimodal generative models [3–5]. With regard to hardware, various acceleration architectures exhibit a growing trend, including central processing unit (CPU) extensions [6–8], graphics processing unit (GPU) and tensor cores [9], and deep learning accelerators [10–12]. The rapid development of software and hardware requires the co-design ability of deep learning systems so that the system can perform multi-level optimizations. In addition, compilation techniques are increasingly being used in deep learning software and hardware systems, which include compiler infrastructures for software and hardware [13,14], deep learning compilers [15–18], and optimizations [19,20]. These compilation technologies provide new co-design opportunities in the golden age of computer architecture and deep learning.

In deep learning hardware–software co-design, hardware architecture innovation is the root, software design is the key, and mapping from software to hardware determines the effect. The deep learning system is responsible for mapping and optimization, whereby the process is divided into model-level optimization, workload partitioning, workload mapping, and hardware interfacing. To achieve higher performance, tuning is required throughout the process, which is essential to co-design. In practice, the most direct method of tuning is to communicate and determine the requirements of the software and hardware teams. In addition, automated co-design approaches, such as auto-tuning, auto-scheduling, and design space exploration (DSE), are also well studied.

With the fragmentation of current deep learning frameworks and hardware platforms, compilation technologies can play a key role in co-design to avoid combination explosions and provide more opportunities in optimization. Compilation technologies include frontend support, intermediate representation (IR), compilation optimization, code analysis, multi-backend code generation, and so on. In the process of software-to-hardware mapping, compilation technologies have been introduced at different levels, such as network architecture search (NAS), model compression, and workload mapping. Most of these levels rely on deep learning compilers [13,15,21,22]. As for the hardware aspect, hardware programming languages [23–26] and compilation infrastructures [14,27] are developing rapidly, to facilitate the design of deep learning accelerators by providing better abstractions. The nature of compilation technologies is the art of abstraction, and abstraction of software and hardware is moving toward a unified IR. We believe that compilation technologies can bring more opportunities to co-design and thus can better achieve the performance and power requirements of deep learning systems.

Many deep learning survey papers have summarized optimizations [19,20], hardware architectures [28–32], co-design approaches [33,34], and compilation techniques [17,18]. To the best of our knowledge, this study is the first to discuss deep learning systems from the perspective of combining compilation technologies and co-design. The contributions of this study are as follows:

- The development of deep learning software, hardware, co-design, and systems is outlined.
- The key technologies of deep learning co-design systems are summarized.
- Compilation technologies in co-design are analyzed for both software and hardware.
- Current problems and future trends of compilation technologies in co-design are discussed.
- A compiler framework for deep learning co-design is proposed.

This paper is organized as follows. The Background section introduces the background of deep learning and co-design development. The Deep Learning System for Co-Design section presents the relationship between deep learning co-design systems and compilation technologies. The Compilation Techno logies in Deep Learning Systems for Co-Design section reviews deep learning compilers and hardware compiler infrastructures. The Current Problems and Future Directions section analyzes the open problems involving compilation technologies and co-design and presents the future direction of our vision. The Buddy Compiler: Deep Learning Compiler Framework for Co-Design section shows the blueprint and progress of our compiler framework for deep learning co-design. Finally, the Summary section concludes the paper.

Background

In recent years, the rapid development of deep learning has increased the demand for improved hardware performance. However, the number of transistors in processors can only grow at a limited rate, indicating that Moore's Law is experiencing a downturn, and Dennard scaling has failed. Consequently, the growth in hardware performance cannot meet the requirements of deep learning workloads. To address this problem, academia and industry have focused on various acceleration architectures that have initiated the golden age of computer architecture [35]. To adapt to deep learning models and hardware architecture innovations, deep learning systems are gradually evolving, whereby compilation technologies are introduced, marking a new era of hardware–software co-design. This section introduces the background on deep learning software, hardware, co-design, and systems.

Development of deep learning software and hardware

The development of software and hardware is complementary for neural networks and deep learning [28]. Figure 1 uses relative frequency data for specific keywords from Google Ngram Viewer to represent trends in artificial intelligence (AI) and hardware innovations. We can conclude from Fig. 1 that improvements in hardware can promote the development of deep learning theory, and workload characteristics can stimulate hardware architecture innovation. This section summarizes hardware and software in relation to the upsurge and decline of neural networks and deep learning throughout history.

Neural networks first emerged in the 1950s and 1960s, but neither hardware capabilities nor software theories were sufficient to support practical applications. At that time, additions were widely used to perform neuronal computation because the multiplication units were too expensive, and thus, primarily binary neurons were affordable for neural network structure. With regard to the theories, "perceptrons" were not linearly separable and therefore could not be used to classify many practical problems, such as the XOR function [36]. The lack of hardware and software support hindered research enthusiasm, which directly led to the first AI winter.

From the 1970s to the early 1980s, general-purpose processors evolved rapidly from unified instruction set architecture (ISA) with complex instruction set computer (CISC) to reduced instruction set computer (RISC), providing an abstraction between hardware and software. Compilation technologies further improved the hardware–software interface, which has led to architecture and programming language innovation. Following Moore's law, the increasing number of integrated transistors allowed for rapid improvement in processor performance.

From the 1980s to mid-1990s, the growth in hardware capabilities promoted theoretical innovation; however, a lack of domain-specific libraries and tools led to bottlenecks. The accumulation of hardware performance over many years led to the development of theories [37–39], thereby opening the second wave of neural networks. Convolutional neural networks (CNNs) achieved initial success in handwritten character recognition and inspired research on hardware accelerators [40–42]. However, problems such as difficulty in collecting data and long training time were attributed to the lack of domain-specific libraries and tools, also raising the possibility that hardware performance was still insufficient. Soon after, the second neural network winter arrived.

In the early 2000s, with the downtrend in Moore's Law, neural network frameworks began to emerge. During this period, instruction-level parallelism (ILP) matured and achieved substantial performance improvements. In the development of neural network structures, tools such as Torch [43] and MATLAB were used. However, the lack of specificity still required much developers' effort, which incurred considerable manual overhead.

In the late 2000s, Dennard scaling began to fade and neural networks began to deepen. Dennard scaling is a statement regarding power consumption, and when it no longer holds, more efficient parallel methods are required. This came about in the multi-core era. Although multi-core architecture did not directly solve the power consumption problem, it further improved performance by introducing more parallelism. However, according to Amdahl's law, the performance ceiling of multi-core architectures depends on the parallelism characteristics of the workloads [44]. Fortunately, the neural network layers became deep [45], and the deep learning era started. The high parallelism characteristics of deep learning workloads have saved multi-core architectures by raising the performance ceiling.

In the early 2010s, the development of GPU triggered rapid growth in deep learning. With years of AI efforts and the compute-unified device architecture (CUDA) ecosystem [46], Nvidia succeeded in deep learning training. Krizhevsky et al. [1] implemented ConvNet with a GPU that outperformed manual approaches, which is a milestone for both CNN and GPU.

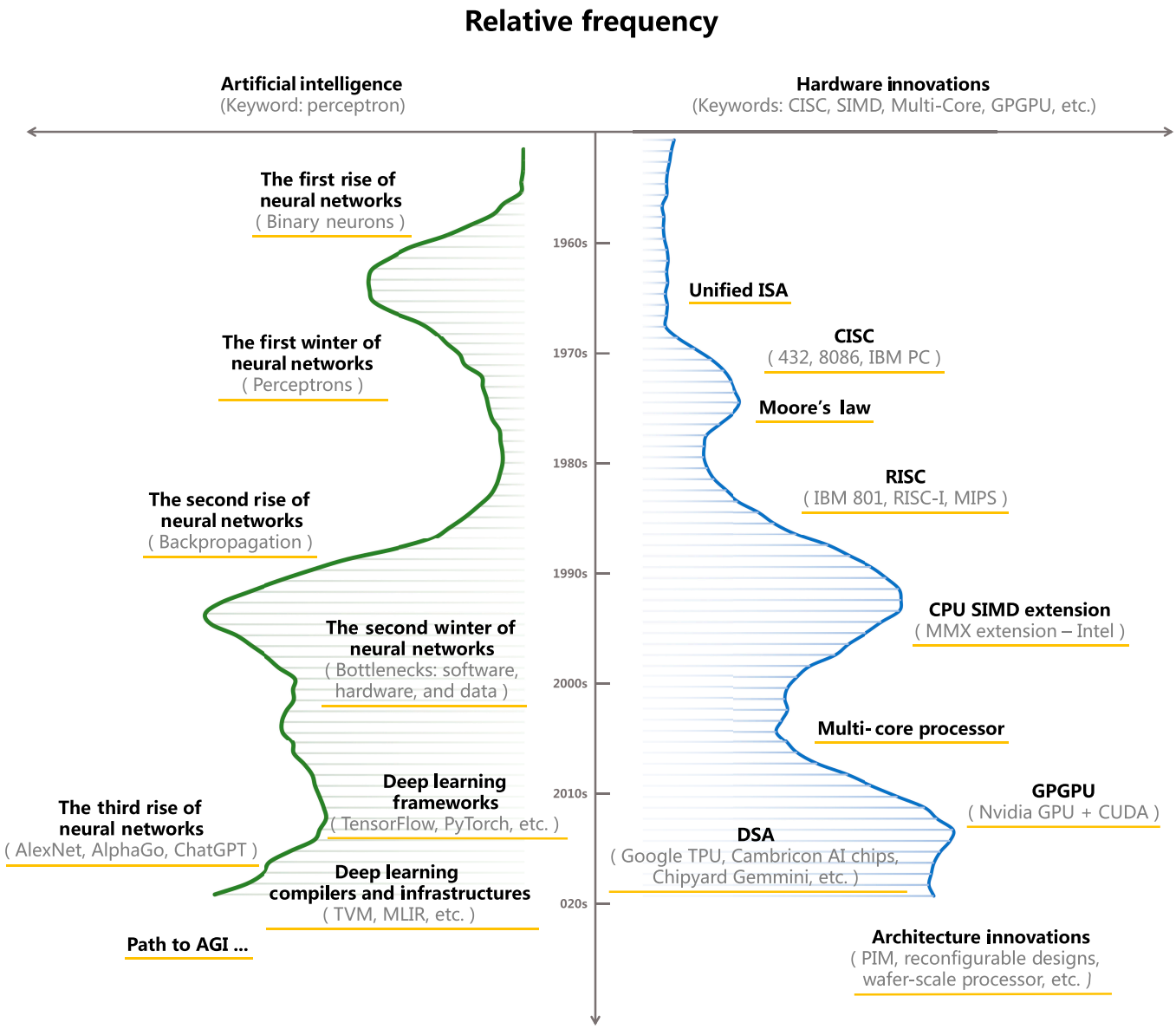

Fig. 1. A timeline of key developments in artificial intelligence and hardware innovation.

Motivated by these results, some deep learning frameworks have also emerged, such as Caffe [47], Chainer [48], and Theano [49]. These frameworks are dedicated to helping developers implement more complex deep neural networks (DNNs) and support GPU acceleration, which has led to several studies on CNN and their variants.

In the late 2010s, domain-specific acceleration architecture (DSA) and deep learning frameworks powered larger neural networks. To achieve higher performance and control power consumption, many DSAs were used for training and inference [10–12,50,51], prompted by the complete end of Dennard scaling. During this time, large tech companies launched open-source deep learning frameworks [52–56] to further facilitate wide application and research. As a result, based on DSA and deep learning frameworks, neural network architectures experienced impressive innovations ranging from RNN to transformers [57,58], whereby the applications were extended to reading comprehension, machine translation, and image generation. However, the deep learning frameworks and hardware tended to be fragmented, so there were many porting requirements, necessitating repeated developments, which increased the manual overhead in the system layer substantially. Consequently, compilation technologies [13,15,21,22,59,60] were introduced to increase reusability and retargetability, providing more optimization opportunities.

Beginning with the early 2020s, large neural networks have brought deep learning to a new peak, placing higher demands on deep learning systems. After iterations of the transformer-based design, generative pre-trained transformer (GPT) series finally achieved great success with ChatGPT. The parameters of these large models have reached the scale of hundred billions, and both training and inference require hardware architectures and clusters with a high degree of parallelism. Various technology companies have introduced AI hardware and platform innovations, such as Nvidia, Xilinx, Intel, Huawei, Cambricon, T-Head, Tenstorrent, Sambanova, BIREN, and so on. Efficient

mapping from workload to hardware is also important; therefore, computational frameworks are developing new techniques to help train and deploy large models. Examples include distributed computing frameworks such as Ray and OneFlow, as well as deep learning platforms like PyTorch 2.0 and OpenXLA, benefiting from efficient compilation techniques. In the foreseeable future, the entire software and hardware ecosystem will move toward an increasingly co-designed system.

Co-design in different periods

Hardware–software co-design is a joint design approach for hardware and software to achieve synergy and meet system constraints, such as performance and power consumption. As shown in Fig. 1, during the early rise and decline of neural networks, software and hardware were developed alternately. By the 2010s, hardware and software for deep learning began to develop rapidly simultaneously, allowing opportunities for wide co-design. The scope of co-design is broad and various approaches have been proposed in different periods and fields. Hardware–software co-design has a history of approximately 30 years, and this section presents the development of co-design over three decades.

The first decade of co-design

In the 1990s, the development of integrated circuit technology led to more integrated hardware designs, and the instruction set processor drove the development of software, allowing co-design [61]. Most of the research focused on the problem of software and hardware partitioning, and the methods ranged from bipartitioning [62,63] to graph-based [64,65] approaches. At the time, the implementation of co-design was relatively inefficient, relying mostly on manual efforts to select the design space and parameters. Subsequently, the emergence of computer-aided design (CAD) reduced the manual cost of co-design, and the development of field programmable gate arrays (FPGAs) blurred the boundaries between software and hardware, providing hardware–software co-design with new directions and opportunities.

The second decade of co-design

In the 2000s, the development of microelectronics powered the integration of heterogeneous system-on-a-chip. Co-design approaches also considered the communication overhead of network-on-a-chip and began to use CAD and multi-objective optimization methods [33]. In addition, DSE initiated the joint design of software and hardware. This led to an increase in the complexity of the co-design system; the required processes include modeling, simulation, optimization, synthesis, and testing.

The third decade of co-design

From the 2010s to the present, the downtrend in Moore's Law and failure of Dennard scaling limited improvement in processor performance and complicated power consumption control [35]. Concurrently, with deep learning on a rapid rise, models became larger, and hardware performance requirements increased. Therefore, various studies on deep learning systems have begun to use co-design approaches. On the software side, the development of deep learning frameworks and high-performance operator libraries exploits hardware acceleration features. On the hardware side, designing DSA can speed up certain workloads and avoid the power-consumption issues of general architectures. In addition, compilation technologies have also entered a golden age with the innovation of hardware architectures and have been widely used in deep learning systems in recent years, leading to new acceleration opportunities with compiler-based co-design [66].

Deep Learning Systems for Co-Design

Deep learning systems can help developers reduce overhead, providing optimization opportunities at different levels. The entire deep learning systems is summarized in Fig. 2, which includes end-to-end deep learning software and hardware systems, hardware development approaches, and co-design techniques. This system can reduce the computational complexity of workloads, enhance execution efficiency, and improve hardware peak performance. This section describes the components of deep learning systems for co-design and the compilation techniques used among them.

Neural network architecture design and optimization

Neural network architecture optimization can reduce the computational complexity of the workload, and the approaches include designing efficient model structures [67–69], pruning [70], quantization [71–74], hardware-aware neural network structure search [75–79], and fusion [15,80,81]. Among these, co-design approaches that use compilation technologies obtain better results. Compared with the general hardware-aware neural NAS, CHaNAS [82] adds compiler-scheduling strategies to the search space to achieve higher performance on the target hardware. MCUNet [83] integrates model design and compiler optimization, which consists of TinyNAS and TinyEngine. TinyNAS searches the neural network architecture, whereas TinyEngine eliminates instruction and memory access redundancies to generate optimized code. For model compression, the compiler pattern can guide structured pruning to reduce the computational complexity of the model, with related works including PCONV [84], PatDNN [85], and CoCoPIE [86].

Workload partitioning and mapping

The end-to-end process of deep learning is to partition the workload into acceleration nodes and further map them to hardware instructions or structures. Partitioning exploits parallelism and improves the scalability of training or inference. Mapping approaches involve both programming models and compiler techniques to optimize workload performance.

Tensor partitioning or model partitioning is for partitioning the input data or entire computational workload to make full use of hardware resources to increase parallelism. Data and model parallelism are commonly used partitioning strategies. Data parallelism is the replication of the model at each acceleration node, allowing computation on different data in parallel with the same model. Model parallelism is the splitting of the model into each acceleration node, with different parts of the model executed in parallel using the same input data. Existing solutions include empirical-based approaches [87] that consider only communication overheads [88], as well as those that consider both communication and computational overheads [89].

Workload mapping transforms coarse-grained operators to instructions or interfaces on the target hardware. Mapping approaches can be divided into deep neural network libraries and deep learning compilers. Deep neural network libraries use programming models designed for hardware to implement high-performance operators [90,91] or reuse of existing basic

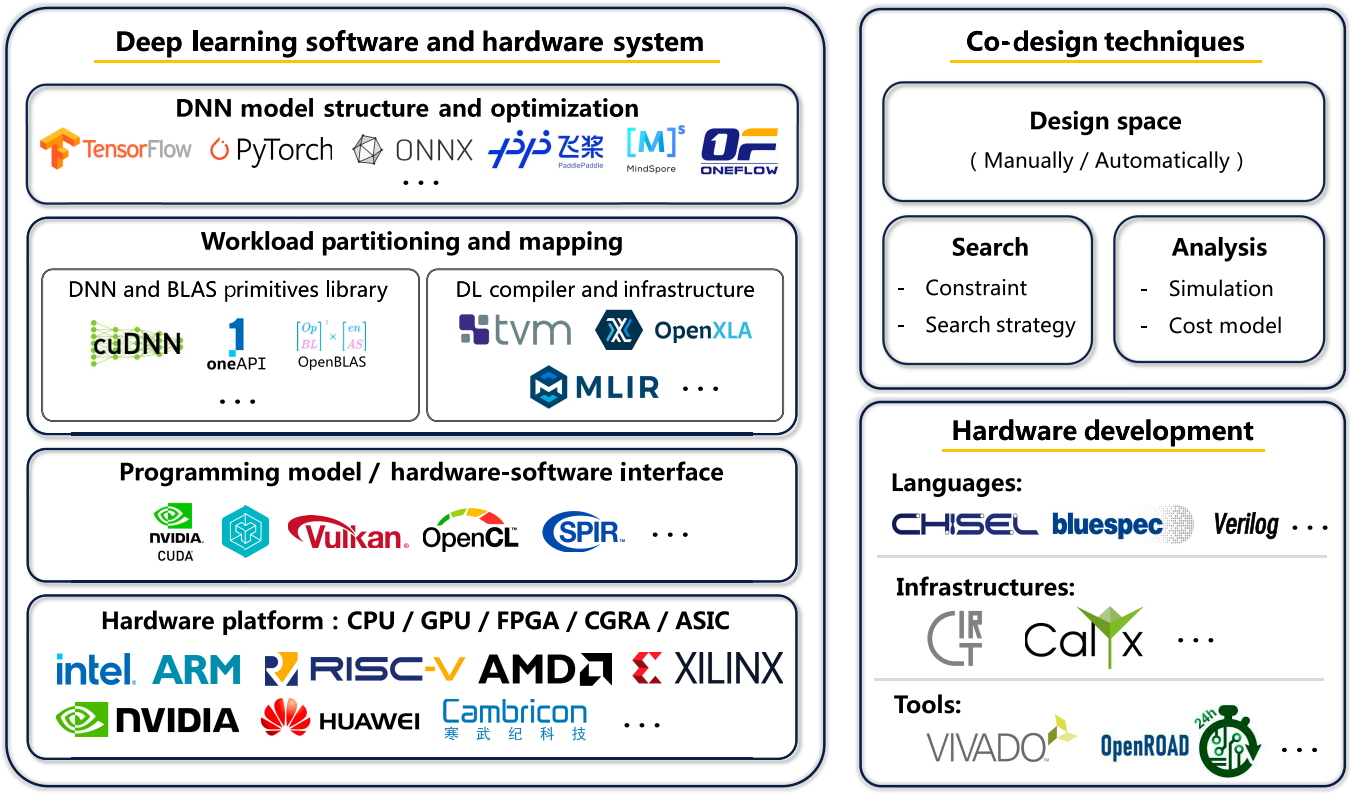

Fig. 2. An overview of deep learning systems.

linear algebra subprogram (BLAS) libraries [92,93]. The development of these libraries relies on manual implementation of optimization algorithms. The hardware compiler maps programming primitives or intrinsic to hardware instructions. Another mapping approach is using deep learning compilers [15,16,21,22,59,60] to translate the deep learning model into an IR, perform multi-level optimization on the IR, and finally use the hardware toolchain for code generation. The use of compilation technologies can increase the reusability of co-design optimization and automation. In addition to these, high-level synthesis (HLS) is also a mapping approach if the backend hardware is an FPGA. HLS directly maps high-level neural network representations to hardware structures for acceleration [94]. The development begins by writing workloads using a specific programming model, such as OpenCL, and advanced compilation techniques are also used to increase the opportunities for HLS optimization [95].

Hardware acceleration categories and approaches

The computing-intensive nature of deep learning workloads and the performance improvement bottleneck of general-purpose processors have jointly led to the development of hardware acceleration capabilities. There are various categories and approaches to hardware acceleration architectures [31,32]. In addition, there are many hardware development approaches [23,24,96] and tools [97,98]. In recent years, some new compilation techniques [14,27,95] have been introduced into the hardware development toolflows. The details will be described in the Compilation technologies in deep learning hardware design section. This section classifies hardware architectures and introduces acceleration approaches.

Classification by hardware platform

CPU acceleration extensions enable data-level parallelism by introducing parallel access memory and compute units. Among these, single-instruction-multiple-data (SIMD) extensions are the most common architectures that can support vectorization of deep learning workloads, whereas emerging vector and advanced matrix extensions are better in supporting deep learning features. For the operators, acceleration can rely on the auto-vectorization capabilities of the compiler or implementation of the optimization algorithm using the intrinsic capabilities provided by the compiler. Some frameworks provide a unified abstract layer for the intrinsic extension of different CPU extensions, usually referred to as a universal intrinsic or global intrinsic layer.

GPUs have powerful thread-level parallelism and can achieve high data throughput, and the well-established hardware and software ecosystem has helped GPUs become the most widely used hardware acceleration architecture in deep learning field. Strategies that have been validated for accelerators are being gradually adopted for GPUs, as evidenced by reduction in precision, support for sparsity, and the tensor core design [29]. The optimization of GPUs mainly depends on the efficient use of multithreading and memory hierarchies [20]. Developers need to implement the optimization strategies by using GPU programming interfaces, such as CUDA, OpenCL, OpenACC, and Vulkan. Some of these have corresponding IRs in multi-level intermediate representation (MLIR) or low-level virtual machine (LLVM) IR for compilation-pass development. In addition, standard portable intermediate representation (SPIR)-V builds an ecosystem for parallel computation and GPU compilers, hoping to become the standard IR for bridging programming interfaces and hardware platforms.

FPGAs offer high performance and low power consumption features. Hardware programmability means that FPGA-based designs are flexible enough to keep up with changes in deep learning workloads. In terms of co-design, FPGAs offer users fine-grained co-design opportunities and greater co-design space, which can further enhance system performance. Optimizations of FPGAs focus on the efficient utilization of numerous digital signal processor units. FPGA vendors also tend to add hardware and software support for deep learning, such as Xilinx's versal adaptive compute acceleration platform [99] and Vitis AI platform [100].

CGRA, which stands for coarse-grained reconfigurable architecture, consists of processing element (PE) arrays and interconnection switches. Its fast reconfiguration ability enables domain-specific flexibility and practicality. The CGRA-based DNN accelerator can be designed as a coprocessor for the CPU to receive instructions and data from the CPU, and the PE executes the statically allocated instructions. In addition, CGRA can also be used as a dataflow processor to accelerate the DNN dataflow graph. These designs require compiler and toolchain support to map deep learning models to specific instructions or graph representations.

ASIC, which stands for application specific integrated circuit, is designed for a specific acceleration architecture that can achieve high performance in power-constrained scenarios. However, this hardware cannot be configured after fabrication. Therefore, most are software programmable to follow changes in deep learning models. ASIC-based design can be a CPU co-processor to accelerate DNN, a stand-alone DNN accelerator, or even a supercomputer. The compiler and toolchain of an ASIC accelerator are often deeply bound to deep learning frameworks, leading to the loss of flexibility in co-design.

Classification by neural network execution strategy

According to their different neural network execution strategies, hardware acceleration architectures can be classified into streaming architecture and computation engine architecture [101]. A streaming architecture is data-driven, mapping the neural network to hardware blocks, where each block represents a layer of the neural network. The blocks of each layer are combined into a pipeline to execute the complete neural network model. Each model has a unique hardware implementation; therefore, this type of architecture is typically implemented on hardware-reconfigurable platforms. A computation engine architecture is instruction-driven such that the compiler maps the neural network to the instructions, and the computation engine is responsible for decoding and execution. The computation engine can be designed as a systolic [10,12] or vector [102] architecture.

Co-design techniques

The previous sections summarize key optimization techniques in deep learning systems, such as optimization of neural network architectures, high-performance libraries and compilers, and hardware acceleration architectures. To achieve the synergistic gains of the above techniques, co-design is an essential step. We believe that the nature of co-design is to balance the trade-off between software and hardware optimization. In practice, software and hardware developers need to understand the design of both sides, communicate efficiently, and make decisions together. Based on our experience, the general co-design process includes the formulation of constraints, the design of interfaces, and the iteration of optimization strategies. After the initial design and implementation, some co-design techniques can be used to fine-tune specific workloads. These techniques include auto-tuning [103], auto-scheduling [104], and DSE [105–111]. The basic idea of these techniques is similar, and the steps include the following: (a) Compose a search space out of parametrized software or hardware design. (b) Select a search strategy to choose a configuration in the search space. (c) Execute the workload using the emulator or hardware according to configured parameters. (d) Collect execution data from the actual hardware or cost model. (e) Refine the design according to these data. (f) Repeat the iterative search until the constraints are satisfied. The search strategy and cost model in the above steps determine the efficiency of the whole tuning process. Some research has already focused on search strategies [112–115] and cost models [103,104,116]. In addition, we think that considering the expansion of the search space by incorporating additional dimensions presents an intriguing direction for exploration.

Compilation Technologies in Deep Learning Systems for Co-Design

Compilation technologies have been gradually introduced in deep learning systems, specifically, deep learning compilers and hardware compilation infrastructures, to provide more opportunities and space for co-design. A traditional compiler can be divided into three parts: the frontend parses the language; the midend optimizes the IR; and the backend generates hardware instructions. The compilation technologies [117–119] in each phase include lexical, syntax, and semantic analyses at the frontend; IR translation, dataflow analysis, control flow analysis, and interprocedural optimization at the midend; and instruction selection and register allocation at the backend. These ideas from traditional compilers can help solve the problems encountered in current deep learning software and hardware systems. This section introduces compilation technologies for deep learning compilers and hardware compilation infrastructure.

Deep learning compilers

Deep learning compilers have evolved from deep learning frameworks. Traditional deep learning frameworks use DNN libraries and runtime engines to execute computational graphs. However, when retargeting multiple hardware platforms, DNN libraries must be reimplemented. The retargetability of the compiler can solve this problem. Therefore, deep learning frameworks introduce the support of compilation technology. There are many stand-alone deep learning compilers compatible with various deep learning frameworks. These deep learning compilers are widely used on the inference side, where the input to the compiler is a pre-trained deep learning model; the frontend converts the model into IR; the midend performs optimization and lowers it to low-level IR; and the backend uses the existing toolchain to generate hardware code. With the emergence of large neural networks, the training framework also needs to introduce compilation technology to achieve acceleration. Recently, PyTorch 2.0 introduced TorchDynamo and TorchInductor to compile PyTorch code into an optimized kernel, using Triton [120] as the programming model for GPU code generation.

The infrastructures of deep learning compilers include frontend importers, high-level language bindings, multi-level IR and its extension mechanism, tensor operations and data structures, optimization passes and a manager, and toolchain integration for hardware platforms. For these, optimization ability and IR design are particularly important, as they determine execution performance and development efficiency, respectively.

Optimization of deep learning compilers enables better mapping of the neural network workload to hardware [110,121,122]. Deep learning compilers typically split the entire DNN model into subgraphs and then apply graph-level optimization technologies to the subgraphs [104], such as layout optimization, operator fusion, and constant folding. Subsequently, the IR is converted to a lower abstraction level for loop-level and hardware-related optimization [103], such as loop reordering, loop tiling, and memory-related optimization.

The evolution of IRs reflects the development direction of deep learning compilers from fragmentation to a unified ecosystem. During IR fragmentation, each compiler defines its own IR and the corresponding frontend, optimization, and backend supports. Tensor comprehension [22] uses HalideIR and polyhedral IR to lower and eventually generate CUDA kernels. Graph Lowering (Glow) [59] takes the dataflow graph from the DNN framework and optimizes it at two IR levels. High-level IR is based on the dataflow graph for high-level optimization and linear algebra representation, whereas lower-level IR is instruction-based for hardware-related optimization; nGraph [60] defines its own nGraph IR and then lowers it to the LLVM IR. nGraph's core components are merged into OpenVINO. Accelerated linear algebra (XLA) [21] transforms TensorFlow graph representations into XLA graph representations and further lowers the XLA graph IR to LLVM IR. The IRs of the above compilers are not reusable, and the optimization must be repeatedly developed. To improve development efficiency, deep learning compilers are on the way of forming a unified ecosystem. Currently, the two major deep learning compiler ecosystems are tensor virtual machine (TVM) [15] and MLIR [13]. The remainder of this section discusses these two ecosystems.

The TVM ecosystem and the MLIR ecosystem

TVM [15] is currently one of the most widely used deep learning compilers, providing an end-to-end approach for compiling neural networks defined by different frontend frameworks to different hardware backends. More importantly, the tensor representation and optimization infrastructure make TVM a reusable ecosystem. TVM converts the DNN model of various deep learning frameworks into a unified IR system, which circumvents the combinatorial explosion problem resulting from multiple frontends and backends. Meanwhile, TVM revisits the Halide idea, which decouples algorithm and schedule, and the powerful auto-tuning and auto-scheduling ability can always provide high-performance solutions. In the ecosystem, automatic kernel generation [123] adds polyhedral transformations to optimize the tensor expression (TE) and supports auto-tuning compatible with the neural processing unit platform. HeteroCL [124] extends TVM to build an FPGA-targeted programming infrastructure comprising Python-based domain-specific language (DSL) and FPGA compilation flow.

MLIR [13] is a multi-level IR infrastructure that is used to build compilers. The idea of MLIR was born in the deep learning field and subsequently developed into a general infrastructure to serve any domain-specific compiler. The extension mechanism and LLVM integration are key to MLIR becoming an ecosystem. MLIR provides core abstractions, called dialects, and powerful extension capabilities. Users can easily add new dialects and define custom operations, types, and attributes. All custom extensions are compatible with the core dialects, and all dialects can be combined to improve reusability. In addition, MLIR is part of the LLVM project [125], and the abstraction level of MLIR is higher than that of LLVM IR. In this case, MLIR can reuse the powerful backend capabilities of LLVM to achieve retargetability. Although MLIR itself does not provide an end-to-end deep learning model compilation process, numerous MLIR-based efforts are under way to form a deep learning ecosystem. The intermediate representation execution environment (IREE) [16] can perform end-to-end compilation and various optimizations, and is regarded as the most innovative work of MLIR. MLIR-HLO [126], Torch-MLIR [127], and ONNX-MLIR [128] connect TensorFlow, PyTorch, and ONNX to the MLIR ecosystem, respectively. Some programming models and domain-specific programming languages have also been incorporated into the MLIR ecosystem. For example, OpenAI's Triton [120] uses MLIR for GPU code generation; xDSL builds a DSL ecosystem based on MLIR; Beaver bridges Elixir and MLIR.

In summary, TVM is an end-to-end compiler for deep learning workloads, and MLIR is a reusable and extensible compilation infrastructure. Although their core ideas differ, the ecosystems built are comparable. TVM's extensibility feature originates from the tensor representation and optimization mechanism and provides a powerful automated design to achieve high performance. However, the domain scalability is relatively limited. MLIR's extensibility feature originates from IR design and LLVM integration. The powerful IR extension mechanism allows a larger ecosystem. However, the specific implementation requires sufficient domain knowledge and experience to achieve the expected result. In the following sections, the components of a deep learning compiler are described in more detail by comparing TVM and IREE. Table 1. summarizes the comparison results.

The compiler frontend

The frontend of a deep learning compiler generates IRs from graph representations of deep learning framework models, which are also known as importers. The converter traverses the computation graph and maps the nodes to IR operations. The frontend converts the models of various deep learning frameworks into a unified IR, thus reducing fragmentation at the framework level. As presented in Table 1, TVM supports more frameworks than IREE, and the essential difference between the TVM and MLIR ecosystems determines the frontend strategy. TVM serves as an end-to-end deep learning compiler solution, whereby the community maintains its own compiler frontend to support deep learning framework conversion into Relay IR. MLIR serves as a compiler infrastructure, delegating frontend support to each deep learning framework. These frameworks provide their own MLIR dialects to directly translate or convert graphs into MLIR core dialects. In this case, IREE only needs to use these dialects or interfaces to connect to the frameworks.

Table. TVM and IREE comparison.

	TVM	IREE
Frontend/Importer	PyTorch, TensorFlow, MXNet, ONNX , Keras, TFLite, CoreML, DarkNet, PaddlePaddle, OneFlow	TensorFlow, TFLite, JAX, PyTorch
IR	• Relay IR (ANF) • TIR (TE + TOPI)	• MLIR core dialects (SSA) • IREE custom dialects • Third-party dialects
Transformation and optimization	• Transformation strategy: transformations are packed as different levels.	• Transformation strategy: transformations are combined as different pass pipelines.
	• Optimization strategy: split algorithm definition and scheduling.	• Optimization strategy: design the algorithm with mixed abstractions.
	• High-level optimizations: operator fusion, layout transformation, etc.	• High-level optimizations: constant expression hoisting, numeric precision reduction, etc.
	• Dataflow analysis: dataflow patterns	• Dataflow analysis: flow dialect
	• Auto-tuning: AutoTVM and AutoScheduler	
Code generation	• VM Bytecode • LLVM IR • BYOC mechanism	• VM Bytecode • LLVM IR • C Code • Other IRs (e.g. SPIR-V)
Runtime and execute mode	• Unified object: PackedFunc • Relay's interpreter • Graph executor • Ahead-of-time compiler • Relay virtual machine • Remote procedure call (RPC) support	• Hardware abstraction layer (HAL) • Virtual machine • Ahead-of-time compiler • Just-in-time engine

IR

IR is one of the most important components of deep learning compilers. It is used to decouple deep learning frameworks and hardware platforms. IR design is also a decisive component of different ecosystems. The IR of a traditional compiler bridges programming languages and backend targets and performs compilation optimization on the IR. Deep learning compilers also rely on the IR of traditional compilers to reuse their existing backend retargeting capabilities. LLVM IR is the most widely used traditional IR. However, LLVM IR has only single-level abstraction and fine-grained operations, making it difficult to map deep learning models with high-level abstraction and coarse-grained operators. Therefore, deep learning compilers always define a high-level IR to address this problem. The IRs at different levels are designed to expand the optimization space and narrow the gap to the low-level IR. Although multi-level IR is not a new compilation technology, it is an important feature of deep learning compilers.

The IR design of TVM can be divided into two levels. Relay IR is responsible for mapping deep learning models and performing graph-level optimization, whereas tensor IR (TIR) performs scheduling and tuning. Relay IR has a functional style with let-binding structure that supports automatic differentiation [129]. In the lowering path, TVM provides a TE to construct TIR, and TE supports various schedule primitives to specify the optimizations (e.g., tiling, vectorization, and parallelization). In addition, TVM uses the tensor operator inventory (TOPI) mechanism to define commonly used tensor operator templates, which can reduce the overhead of the manual approach using TE. After scheduling and tuning, the generated TIR is further translated into LLVM IR.

MLIR is the IR infrastructure of IREE with a multi-level structure that uses the concept of a dialect to represent an abstract level, and each dialect includes operations, types, and attributes. The IRs in the IREE project can be divided into MLIR core dialect, IREE custom dialect, and third-party dialect. As infrastructure, MLIR develops many mechanisms to help users easily design IR. It provides an operation definition specification (ODS) framework for IR definition and a table-driven declarative rewrite rule (DRR) for pass writing. In addition, studies such as IRDL [130] further simplify the workflow. MLIR also defines some core dialects as general abstractions to increase reusability, and IREE includes custom dialects such as flow, hardware abstraction layer (HAL), and virtual machine (VM) as extensions. To connect with various deep learning frameworks, IREE reuses third-party dialects, such as the Torch dialect of Torch-MLIR and the HLO dialects of TensorFlow. The compiler gradually lowers the model-level dialect to the LLVM dialect and finally generates LLVM IR.

TVM IR design and MLIR share similar ideas, but there are differences in their principles and systems. The essential difference lies in the structure of the IR. Although both represent the dataflow of the deep learning model, the a-normal form (ANF) IR structure used by TVM is different from the static single-assignment (SSA) IR structure of MLIR. An ANF-based design can easily express functional semantics and specify the scope of the computation, thus eliminating ambiguity in semantics [131]. The SSA-based design can easily perform optimizations such as constant propagation and dead code elimination [118]. From a systems perspective, the fundamental difference between MLIR and TVM determines the characteristics of IR. As an end-to-end deep learning compiler, TVM provides a highly integrated IR design, and the attached scheduling and tuning make it easy for compiler users to get started, but difficult for compiler designers to extend. As an infrastructure, MLIR provides a modular IR design, and the definition and extension tools substantially reduce the engineering cost of compiler designers. However, compiler users need to have sufficient experience to deal with various passes.

Transformation and optimization

A deep learning compiler can perform transformation and optimization to improve model execution efficiency. At the graph level, optimization approaches primarily include fusion and data layout transformation in the high-level abstraction. Fusion can reduce memory access and the communication overhead of the intermediate data, and data layout transformation can generate a backend-friendly data format. After further lowering, peephole optimization based on sliding windows and global optimization based on dataflow analysis can be performed [117]. Concurrently, many specific optimization algorithms can be used to exploit hardware acceleration architectures. CPU-oriented optimization approaches primarily include vectorization for SIMD/vector extension, parallel computing for multi-core architectures, and memory access optimization for memory hierarchy. Although GPUs do not have the same large cache and high frequency as CPUs, their high-throughput and parallel structure help accelerate deep learning workloads. Therefore, GPU-oriented acceleration approaches typically adopt multithreading and memory access optimization [20]. For different architectures, auto-tuning techniques can increase the effectiveness of automatic compiler optimization. The optimization strategies parameterized by the compiler form a search space, and the target machine or cost model reports the results of each search step. The search process is iterated until the target constraint is satisfied. In addition to relying on compiler optimization, programmers can manually optimize code using a hardware programming model. For example, an intrinsic model or programming model (e.g., CUDA, OpenCL, and Vulkan) can be used to write optimized algorithms for CPU extensions or GPU. In this case, the role of the compiler is to support various levels of abstraction and map the intrinsic or programming model. A previous survey provided an extensive summary of various optimization techniques [17], and the compiler infrastructure for implementing optimizations is discussed below.

At the IR level, both TVM and IREE can achieve the above optimizations. However, they differ in their implementation of the optimization strategy. TVM splits the definition and scheduling of the algorithm, which can increase the readability of the algorithm and allow the effective use of the scheduling representation for tuning. IREE uses MLIR infrastructure to add custom dialects and implement optimization passes with mixed abstractions. The optimization strategies of both TVM and IREE are implemented as passes in the compiler, and each pass is an IR-level transformation. The pass infrastructures have similar components for managing the definition, registration, and execution of passes. The differences in the implementation details arise from different IR designs. MLIR introduces a declarative definition mechanism that allows passes to support flexible and extensible IR. Multi-level abstraction design, coupled with the usage strategy of mixed dialects, presents challenges in MLIR pass pipeline arrangement. Some high-level optimization passes across multiple dialects require careful arrangement of passes to achieve the expected effect. The Relay IR and TIR abstraction partitions of TVM facilitate the construction of dependencies between passes and package optimization at different levels. In addition, TVM has powerful auto-tuning capabilities. AutoTVM [103] requires a search space and iteration definition, thereby adjusting the optimization parameters in each search step to generate code according to the selected configurations. To perform more efficient auto-tuning, Ansor [104], known as AutoScheduler, was proposed as an advanced approach that does not require manual specification of the search space.

Code generation

The code generation process of the traditional compiler is to convert IR into the assembly code of the target platform, and the key processes are instruction selection and register allocation. Instruction scheduling can be performed before and after register allocation, generating a high-quality code to exploit ILP. The deep learning compiler executes the model, generates corresponding code according to different execution methods, and feeds the generated code to the processor or executor. In this section, code generation is classified into machine code generation and engine instruction execution. The two types correspond to domain accelerator building approaches (i.e., domain-specific instructions and engines) [29]. Domain-specific instructions are extensions of general-purpose instructions that drive specific hardware and accelerate the workload. This approach requires a compiler to wrap instructions as corresponding programming interfaces, whereby the programming interfaces are used to implement high-performance operators. In addition to exposing the programming

interface, the optimization pass in the compiler can map workloads to hardware. Regarding the release of the instructions, the deep learning compiler can be regarded as a domain extension of the traditional compiler. Thus, the infrastructure can be reused during the code generation process. However, domain-specific engines must implement compilers to generate graph representations or bytecodes for custom drivers, executors, and VMs.

The results of TVM code generation are encapsulated into unified runtime objects. The actual generated code objects are determined by the execution method. TVM's graph executor and VM require compilers to generate graph representations and bytecodes, respectively. An alternative is to use an ahead-of-time (AOT) compiler to generate shared libraries. Furthermore, TVM provides a bring-your-own-codegen mechanism for backend providers to register their generated custom code as the backend of the relay compiler.

IREE uses a VM to execute the program by default; therefore, the high-level IR generates the VM bytecode after gradual lowering. IREE's HAL dialect provides a series of operations for code generation to determine the dispatch region, application binary interface (ABI), and definition of input and output. If static deployment is required, IREE can generate LLVM IR, C code, and other IRs from the MLIR representation. Taking the GPU target platform as an example, IREE can generate NVVM code through MLIR and LLVM IR, or convert it to the SPIR-V dialect and then generate the SPIR-V binary.

Runtime and execution mode

The IR abstracts the structure of the deep learning model, and the code generation process translates the IRs into executable files. To run these executables on the target hardware platform, it is necessary to maintain a runtime environment to interact with the operating system and other system software. For runtime support, the compiler needs to know the specifications of memory allocation, visibility rules, procedure calls, ABI, etc. These conventions ensure that the different software can interoperate. Deep learning compilers must also consider heterogeneous architectures and their runtimes. The runtime of the heterogeneous architecture bridges the host program and kernel. It can also build the kernel, set kernel parameters, load the kernel, and start it.

Similar to program execution, deep learning models can be classified into AOT compilation, interpretive execution, and just-in-time (JIT) compilation. AOT compilation converts the deep learning model into machine code and links other programs into an executable file that can run directly on the target machine. Interpretive execution compiles the model not into machine code but into an intermediate state such as an abstract syntax tree (AST), graph representation, or bytecode, and then uses an interpreter or VM for execution. JIT compilation combines the features of AOT compilation and interpretive execution, compiling the IR into machine code at runtime, which can be further optimized based on runtime information, balancing performance, and flexibility.

In the TVM runtime environment, runtime objects' output during code generation can be exported, loaded, and executed. The design of TVM runtime adopts a modular strategy, and developers can add new data structures at runtime. TVM encapsulates the execution mode as the interface of the runtime object, where the execution modes comprise the following categories: (a) The relay interpreter executes a program by traversing the AST, which is inefficient and unsuitable for deployment. (b) The relay executor runs the graph representation, which makes good use of static information to optimize memory allocation. (c) Relay AOT compilation allows local execution by building a relay into a shared library, which leads to high performance, but is difficult to extend and modify. (d) The relay VM provides a dynamic execution environment to run the bytecode, which can support dynamic shape, balance performance, and offer flexibility. (e) TVM remote procedure calls allow deployment on remote devices.

The runtime environment of IREE mainly comprises a HAL and VM. A HAL provides a consistent interface across execution targets, enabling executable definitions, device memory allocation, command-buffer assignment, device communication, and synchronization. The VM is responsible for executing the bytecode generated by the code-generation process. The program is loaded into the command buffer and dispatched to a hardware platform through the HAL interface. IREE's execution model introduces timeline and fence components of timeline to support flexible invocation and scheduling, such as sequential execution, pipeline execution, and stream-ordered allocations. In addition to using the VM execution method, IREE can perform AOT and JIT execution when integrating different frameworks.

Compilation technologies in deep learning hardware design

In addition to deep learning compilers, innovations in compilation technologies benefit hardware design. With the rapid development of deep learning hardware, hardware design has exposed language and abstraction problems [14,132]. Compilation technologies can provide a higher level of abstraction and unified support for fragmented languages and tools. This section focuses on compilation technologies for deep learning hardware designs.

Compilation methods for different hardware architectures

According to the Classification by neural network execution strategy section, deep learning hardware architecture is divided into streaming architecture and computational engine architecture. These two architectures use different compilation methods to map the neural network workload to the hardware [101,133].

Streaming architecture is data-driven. The compiler can analyze the neural network and hardware resources, thereby mapping the computation graph to a specific hardware description. In this process, the compiler obtains sufficient information to perform the optimization. As for the hardware design, HLS is commonly used to generate streaming architectures. Computation engine architecture is instruction-driven. Hardware design uses hardware languages to implement architectural templates. Compilers are responsible for mapping languages to low-level representations, such as Verilog [96]. Some hardware designs use HLS to design interfaces and control logic, whereas others use hardware languages to design computing units. The template of the hardware architecture can be instantiated with an appropriate configuration based on the neural network and hardware resources.

Compilation technologies in HLS

HLS increases the abstraction level of the hardware design to the level of C/C++/OpenCL [134–136]. The compiler uses heuristic rules to map high-level representations onto hardware description language (HDL) [137,138]. Programmers must add

a directive (i.e., pragma) to guide transformations, which typically rely on solvers to satisfy various constraints [139,140]. Compared with low-level HDL, HLS can accelerate the design process and improve development efficiency.

The compilation tools of HLS use different optimization strategies to map high-level representations to efficient RTL code. Vivado HLS/Vitis HLS supports the compilation and optimization of the C program to RTL code. The optimization methods depend on the pragma in the program, including dataflow design and pipeline arrangement, as well as unroll and tiling strategies. Xilinx's SDSoC [141] further raises the abstraction level, and the compiler converts the C/C++ program into a complete software and hardware system that allows hardware acceleration on selected parts. The tool analyzes the selected part and identifies the calculation-intensive area, optimizes the hardware code using the Vivado HLS tool, arranges the data transfer between the CPU and acceleration architecture, and reorganizes the data access pattern between the processing system (PS) and programmable logic (PL). The Intel OpenCL SDK [142] provides a set of compilation tools for converting the OpenCL programming model into an FPGA-oriented hardware design. Many optimizations can be performed during compilation. For example, compilation tools can eliminate redundant hardware resource requirements, unroll and vectorize loops, and generate multiple computational units for each kernel.

The aforementioned HLS compilation tools require users to manually write primitives to guide hardware code generation or specify hardware acceleration areas. To achieve an efficient design, users must have sufficient hardware domain knowledge and be familiar with specific programming models and primitives. In addition, these compilation tools use AST or IR as representations to analyze and optimize the input C/C++/OpenCL. The abstractions are single-level and oriented toward software logic and are not conducive to representing high-level hardware information [101]. Therefore, many optimization opportunities are lost during the compilation process. To address these issues, ScaleHLS [95] adds multiple levels of abstraction to existing HLS tools and performs optimization at multiple levels. ScaleHLS is dedicated to providing an extensible and customizable HLS framework. Regarding the IR design, ScaleHLS uses MLIR as the compilation infrastructure. Specifically, the multiple levels of IR include graph, loop, and directive levels. Analysis and optimization can be performed at the corresponding levels. ScaleHLS analyzes the dependencies of dataflow nodes at the graph level, conducts reordering and tiling at the loop level, and performs pipeline arrangement and array partitioning at the directive level. In addition, ScaleHLS supports DSE and code generation for the Vivado HLS tools. Ultimately, ScaleHLS can achieve substantial performance improvement over traditional HLS compilation tools.

Compilation technologies for high-level hardware language

High-level synthesis raises the level of hardware design abstraction. However, traditional HLS languages and tools use a black-box strategy for mapping to hardware, making the final result unpredictable. To address such issues, high-level HLS languages and tools have been proposed, to clarify hardware implementation and generate predictable architectures [26]. In addition to HLS languages, high-level HDLs can better describe predictable processor and accelerator architectures [23,24]. This section discusses compilation technologies that support these high-level hardware languages.

Dahlia [26] proposed a general-purpose hardware language with a higher level of abstraction than traditional HLS languages, proposing a system that can predictably compile HLS programs into hardware accelerators. The compiler can convert Dahlia into HLS C++, and the generated code is compatible with existing commercial HLS toolchains. During the compilation process, the Dahlia compiler can perform type checking to avoid memory-type conflicts, parallelize and reorder code without dependencies, and execute unrolled copies in parallel. In the future, it is expected that compilers will directly generate RTL-level code, thus completely eliminating the unpredictable drawbacks of existing HLS tools. The language can be further simplified to reduce the cost of the compiler implementation.

Bluespec [23] is a high-level HDL. It follows the idea of Haskell in designing a powerful abstraction for describing high-level and parameterized architectures. Bluespec builds on the basis of traditional hardware semantics and uses software strategies for static verification and elaboration. Bluespec's compiler *bsc* [143] compiles the high-level HDL into a synthesizable Verilog, and then the RTL tool can use the generated code for a standard process. The distinct feature of *bsc* lies in the production of a synchronous, clocked Verilog with control logic that preserves atomic-transaction behavioral semantics, which is referred to as rule scheduling. In addition, *bsc* can optimize control logic and analyze the relationships between rule conditions using the SAT solver. In compilation mode, *bsc* supports separate package compilations to achieve incremental rebuilding. Bluespec's commercial compiler and library have been widely used since 2000, and the open-source version has been available since 2020.

Chisel [24] is a general high-level hardware programming language embedded in Scala. Chisel can easily design low-level hardware blocks and extend them to high-level hardware-design patterns. The generated RTL-level code is compatible with the standard processes of FPGA and ASIC. A high-level language supports the design of parametric hardware generators, overcoming the difficulties of traditional HDL. Chisel uses flexible intermediate representation for RTL (FIRRTL) [144] as the IR and compilation framework. FIRRTL includes direct semantics, simple nodes, and user-friendly interfaces. These characteristics help Chisel conduct better transformations and optimization. The transformations are performed on the AST, and optimizations include constant propagation, common subexpression elimination, and dead code elimination. Based on Chisel and FIRRTL, Gemmini [12] proposed a full-stack DNN accelerator generator. In terms of performance, the accelerator generated by Gemmini is greatly improved compared to state-of-the-art methods. Gemmini uses the processor cores of RocketChip [145] and provides an opportunity for system-accelerator co-design. This also demonstrates that a general-purpose high-level hardware language combined with a reusable hardware compilation framework can achieve high-performance hardware design and reduce development and verification costs.

Spatial [25] is a high-level DSL that is used to implement application accelerators. The template-based design allows spatial balancing of programmer productivity and design performance. The spatial IR is a hierarchical dataflow graph, and the nodes in the graph include control, memory, and primitive nodes. The compiler translates the Spatial program into a

synthesizable Chisel representation that can support FPGA and Plasticine CGRA [146]. The compilation process includes pipeline scheduling, automatic memory banking, and automated design tuning. Spatial relies on these parameterized passes to perform DSE. The RTL templates written in Chisel help instantiate the hardware module in the code generation phase.

Hardware compilation infrastructures

Hardware compilation infrastructures serve as the basis for hardware language compilers. Infrastructures unify abstraction and increase the reusability of hardware design compilers. The components include unified IRs, extensible mechanisms, and reusable optimization. This section discusses the recently proposed infrastructures Calyx [14] and CIRCT [27], presenting their design ideas and practical applications.

Calyx [14] is an intermediate language (IL) that serves the compilation process from high-level programs to hardware designs. Calyx's IL combines imperative and structural sublanguages to describe the control flow, hardware modules, and their connections. In addition, Calyx provides a compiler infrastructure for hardware design and existing DSL support. The compilation process includes code analysis and optimization, and the code generation phase produces synthesizable RTL code. In terms of optimization, Calyx provides resource sharing, register sharing, and cycle latency inference. As a compilation infrastructure, Calyx allows hardware designers to use high-level and user-friendly languages and helps language designers create a unified IL and reusable compilation pipelines. Calyx-based hardware design has a lower implementation cost than HLS and can outperform HLS approaches in terms of performance.

CIRCT [27] is an open-source modular hardware compilation infrastructure that unifies the abstraction of various hardware design tools at the IR level by applying the design methods of MLIR [13] and LLVM [125]. CIRCT focuses on the fragmentation problem of abstraction and tools in hardware design. The CIRCT community believes that the core of heterogeneous system design lies in the representation and manipulation of abstractions. CIRCT inherits the characteristics and infrastructure of MLIR, which allows CIRCT to integrate existing toolflows, define new abstractions, reuse optimizations, and generate code. CIRCT can support multiple IRs, including MLIR, FIRRTL, and Calyx. In the compilation pipeline, CIRCT converts these into core hardware dialects and generates the RTL code. It is rapidly evolving and is expected to drive the development of open-hardware design tools.

Current Problems and Future Directions

The previous sections analyzed deep learning co-design systems and related compilation technologies. An evolving trend is observed from fragmentation to ecosystems, and this section summarizes the current problems and proposes future research directions.

Problems and directions of co-design frameworks

Problem: Lack of a unified co-design framework

With the development of large neural networks, AI training computation requirements are growing exponentially with the rate doubling every 3.4 months [147]. This goes far beyond Moore's Law, even in its heyday. Therefore, we believe that only an in-depth co-design can satisfy the performance requirements.

The key to co-design is to enable software and hardware designers to communicate and collaborate efficiently. However, there is currently no unified framework that can assist in co-design; therefore, various hardware–software co-design works and tools are not easy to use. There is a gap between software and hardware design methods and ideas. In principle, co-design can coordinate software and hardware designs to improve performance and efficiency. However, co-design is difficult in practice because of the lack of a common infrastructure. In the current approach, algorithms, programming languages, compilers, and hardware infrastructures are designed independently, and it is difficult to achieve the synergistic effect of co-design. We believe that framework-level work is often based on unified abstraction. For example, deep learning frameworks rely on a unified computational graph, and compiler frameworks rely on a unified IR. Therefore, the lack of a unified software and hardware abstraction is the cause for the lack of co-design frameworks.

Direction: A unified abstraction for co-design

A unified IR for software and hardware is key to implementing DSA and supporting programmability [27]. Software programming models are developing toward higher abstraction levels and domain-specific directions. Hardware architectures are becoming parallelized, heterogeneous, and customized. In co-design, a unified software and hardware abstraction can form a larger design space and facilitate software and hardware partitioning at the system level. This abstraction would also allow software and hardware developers to better communicate and understand each other. Currently, MLIR [13] is used as the unified IR for software and hardware. Software IR can be used to map high-level abstraction to hardware instructions, and hardware IR can support the design of HLS or accelerators. However, both MLIR and CIRCT are compiler infrastructures in their early experimental stages. We believe that a unified IR for software and hardware will be a future research direction. A co-design framework based on this type of IR can form a software and hardware ecosystem to solve the fragmentation problem in software and hardware design.

Problems and directions of multi-layer IR compilation infrastructure

Problem: Efficiency issues with IR-to-hardware mapping

Multi-layer IR has been proven to be a necessary component of deep learning compilation technology [13,15]. Different levels of abstraction can be connected to existing software and hardware to improve reusability. These IR designs create new optimization spaces and allow the reuse of existing optimizations. A previous survey paper also mentioned some directions for compilation optimization [17], and some recent studies have explored various features, such as support for dynamic shapes [148], sparsity [149,150], quantization [151], and CPU/GPU optimization [152,153]. However, these advantages come at the cost of compilation efficiency issues. It is obvious that the increase in the number of abstraction layers leads to a decrease in compilation speed. In addition, sufficient experience is required to choose multiple passes and confirm multi-layer compilation pipelines to achieve collaborative optimization effects. Most developers are not proficient

in all passes, which also means that development efficiency is reduced, and the execution efficiency of the program is not optimal.

Direction: Efficiency optimization of multi-layer compilation

The efficiency of multi-layer compilation can be approached from two aspects: development efficiency and execution efficiency. In our opinion, improving compilation efficiency using automated approaches will be a future research direction. From the developer's perspective, the compilation pipeline of multi-layer IR needs to be manually specified, which is the main reason for its low development efficiency. If the compiler can analyze the source program and automatically arrange the pass pipeline, development efficiency can be greatly improved. In addition, manually specifying the optimization passes and related configuration may not maximize execution efficiency. On the one hand, the execution efficiency of the passes themselves can be improved by automatic parallelization. On the other hand, adding auto-tuning support to the pipeline configuration mechanism can ensure considerable execution efficiency on the target hardware. The efficiencies of these automated approaches should also be considered. As the number and complexity of the abstractions increase, the search space for the automation mechanism expands geometrically. Therefore, design space reduction and search acceleration for multi-layer compilers are future research directions.

Problems and directions of domain-specific compilation technology

Problem: High cost of domain-specific compiler implementations

Deep learning compilers, as important domain-specific implementations, can drive the development of domain-specific compilation technologies and also amplify some of the problems. Domain-specific language and architecture researchers, in addition to focusing on the design of the programming language and hardware architecture, also need to spend a lot of time implementing the entire compiler stack to support end-to-end tasks. On the programming language side, most languages are C/C++ or Python-embedded designs, and there are also some stand-alone language designs that require the modification of existing compiler frontends and language-specific bindings. On the hardware side, especially for instruction-driven architectures, a large amount of engineering is required to complete the mapping from software abstraction to specific instruction sets. Although there are some frameworks and tools that can help reduce the development burden arising from the uncertainty of the frontend language and backend instruction set, the compiler development of the frontend and backend is difficult to reuse and requires repeated efforts.

Direction: Extension mechanisms for domain-specific compilers

The overhead of domain-specific compiler implementation originates from the wide variety of frontend languages and backend instruction sets. In the future, it will be necessary to unify frontend and backend standard techniques and add domain-specific extension mechanisms. Python is a programming language well-recognized in deep learning, but the nature of the language makes it unsuitable for efficient hardware mapping [35]. While Python's position and ecosystem in the deep learning field seem unassailable, it would be interesting to conduct research on stand-alone deep learning languages from a programming language perspective. If the frontend language standard can be determined, regardless of whether it is a Python-based or a stand-alone language, designing a unified extension mechanism based on this language can substantially reduce the development overhead of the frontend compiler. For the backend, RISC-V is currently the most suitable instruction-set architecture for domain-specific hardware because of its modular and extensible features. If the compilation infrastructure could support RISC-V features, it would greatly reduce the overhead of implementing the compiler backend. However, most current compiler frameworks do not support flexible and extensible mechanisms, requiring intrusive modifications of existing compilers. Therefore, building a flexible and scalable compiler backend framework for RISC-V could solve the problem of domain-specific instruction set support.

Problems and directions of the co-design compilation ecosystem

Problem: Implicit fragmentation in the unified ecosystem

With the development of the TVM and MLIR ecosystems, many existing frameworks are gradually connecting these unified ecosystems. This trend can increase reusability and achieve synergy, but implicit fragmentations remain in a unified ecosystem. The purpose of forming an ecosystem is to share infrastructure and reduce implementation overhead. However, some infrastructures are still in the process of rapid development and have not yet reached a stable version, leading to frequent synchronization and high maintenance costs. Projects with different versions cannot be reused or integrated because of interface incompatibility. More importantly, repeated definitions at the same level of abstraction also cause implicit fragmentation. For example, some studies redevelop high-level abstractions to carry specific frameworks or hardware information, which can lead to framework-level fragmentation. These implicit fragmentations undermine the expected generic and reusable features of a co-design ecosystem.

Direction: Better infrastructures for the co-design compilation ecosystem

The implicit fragmentation problem in unified ecosystems arises from imperfections in the infrastructure. We believe that the infrastructure in the ecosystem should add a compatible modification mechanism for core abstraction, a cross-project registration mechanism for optimization passes, and HALs for code generation. The compatible core abstraction modification mechanism can reduce the repeated implementation of the same abstraction level. However, it is difficult to make the compiler compatible with default and modified operations, which may require a non-intrusive modification approach to add more field information to the operations. The joint use of optimization passes in different projects is difficult mainly because of the lack of a cross-project pass registration mechanism. In addition, a stable upstream version is required to ensure consistent interfaces. The HAL can reduce the development of specific hardware and increase code portability. The difficulty of the HAL is in balancing the abstraction level and performance to achieve maximum performance for a particular hardware while maintaining a high abstraction level.

Buddy Compiler: A Deep Learning Compiler Framework for Co-Design

The previous sections reviewed compilation technologies in deep learning co-design systems and discussed the current problems and future directions. Based on the above analysis, we propose a domain-specific co-design compilation framework called the Buddy Compiler. Our framework aims to lower both software and hardware to the IR level for joint optimization, to allow compiler-architecture co-design. On this technical route, we embrace MLIR and RISC-V to form a modular and extensible hardware–software co-design ecosystem extending from DSL to DSA. As shown in Fig. 3, our framework is divided into five modules: compiler framework, benchmark framework, DSA platform, co-design module, and compiler as a service platform.

- The compiler framework (buddy-mlir) can help developers easily design and implement a domain-specific compiler.
- The benchmark framework (buddy-benchmark) can help developers evaluate the performance of different case levels.
- The DSA platform (buddy-dsa) provides accelerator templates and compiles high-level hardware descriptions at the IR level.
- The co-design module (buddy-codesign) enables joint auto-tuning or DSE for both software and hardware at the IR level.
- The compiler as a service platform (buddy-caas) can be used as the ecosystem entry, allowing the use of the proposed infrastructure online.

Among the above modules, both buddy-mlir and buddy-benchmark are already open source projects that can support end-to-end compilation and performance evaluation. Additionally, buddy-caas has been used online for configuring MLIR pass pipelines and integrating RISC-V backends. The remaining two modules, buddy-dsa and buddy-codesign, are being designed and developed and will be open source in the future.

Buddy Compiler overview

As a compiler framework, Buddy Compiler is committed to building a scalable and flexible hardware and software co-design ecosystem based on MLIR and RISC-V. Its design strategically employs modular implementation, decoupled infrastructure, and a user-friendly toolchain to solve the fragmentation and dependency problems of the current co-design software and hardware technology stack. This section introduces the Buddy Compiler modules.

Compiler module

The compiler module (buddy-mlir) is built around MLIR and is dedicated to providing compilation support for DSL to DSA. This module is expected to serve researchers and engineers with programming languages, compilers, and DSA backends.

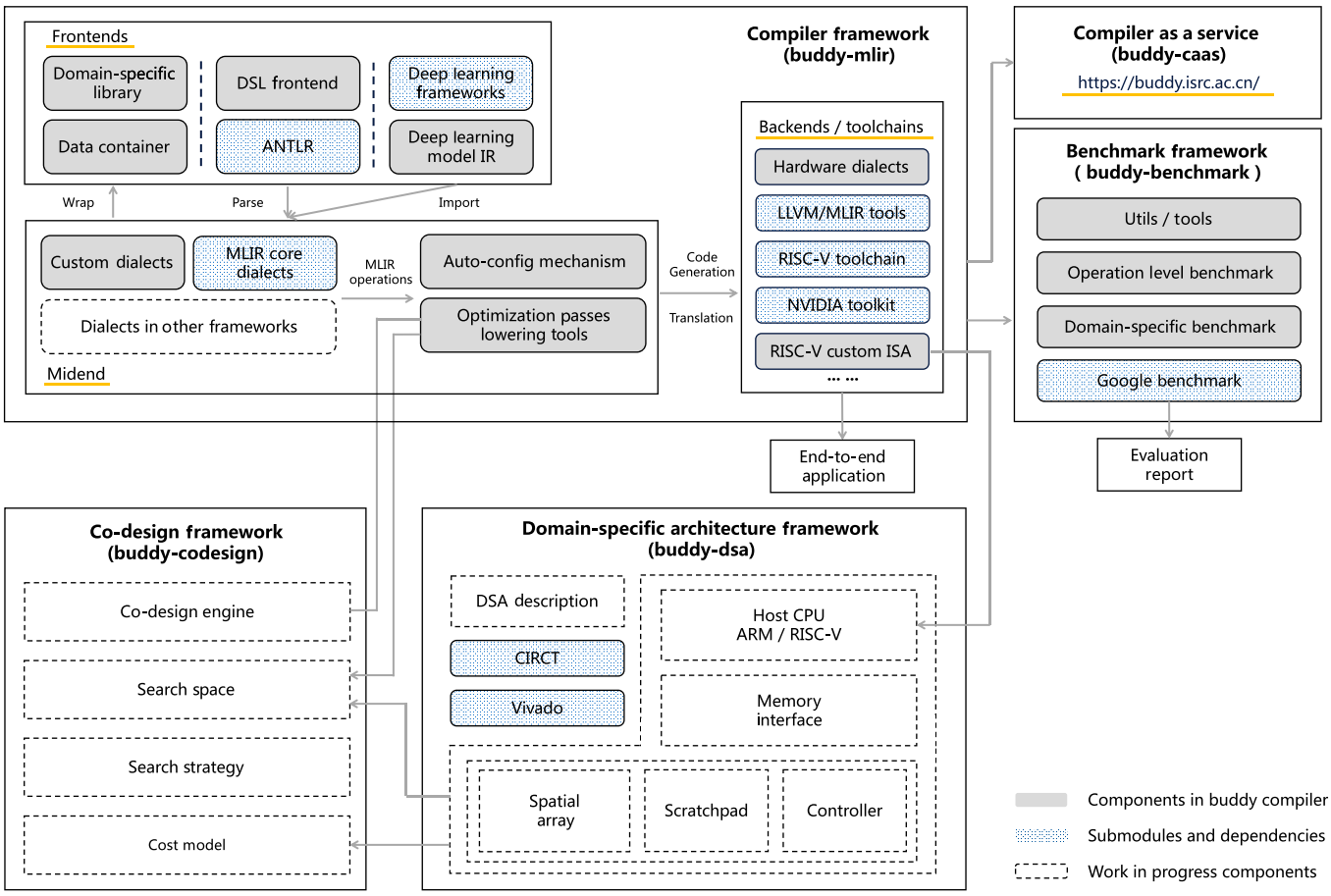

Fig. 3. Overview of the buddy compiler.

Regarding DSL support, language bindings have mature implementation methods, particularly Python bindings. We use the MLIR Python binding to interface to some deep learning frameworks, such as PyTorch 2.0. In addition, our frontend also supports stand-alone DSL by implementing DSL to MLIR mapping based on Antlr. This reduces compiler development overhead for DSL designers. The midend adds custom MLIR dialects to support domain-specific applications or hardware. We used a combination of dialects at different levels to implement the optimization algorithms as passes. The Auto-Config mechanism is designed to automatically configure optimization passes based on the hardware platform, enabling hardware-aware optimized IR generation. With the powerful retargeting capabilities of MLIR and LLVM, the backend can support many toolchains. However, it is not convenient to support domain-specific ISA extensions. Although the LLVM backend infrastructure can be reused with RISC-V as the instruction set, this requires intrusive source code changes that make upstream synchronization and independent extensions difficult. Therefore, we implemented an out-of-tree instruction extension method in buddy-mlir that guarantees both extensibility and synchronizability. In the future, more automated methods based on the current end-to-end compilation process will be introduced to further improve development efficiency.

Benchmark module

The benchmark module is dedicated to providing multi-level performance evaluation. On the deep learning side, we provide the following capabilities. (a) Model-level evaluation, such as ResNet and MobileNet. (b) Operation-level evaluation, such as convolution and GEMM. (c) Preprocessing-level evaluation, such as digital image processing and digital audio processing. (d) Compilation optimization-level evaluation, such as auto-vectorization.

The benchmark cases for each level can be expanded, and new levels and fields can be added. We currently rely on Google Benchmark for performance reporting, which provides basic execution time metrics. In the future, more metrics and simulator-based evaluations will be added for DSA.

DSA module

The DSA module is dedicated to the design and implementation of RISC-V-based programmable accelerators and provides interfaces for hardware–software co-design. In terms of programmability, the RISC-V ecosystem is currently the best choice because of its modular and extensible design concept, and the buddy-mlir module supports compilation for custom RISC-V instruction sets. From the co-design perspective, the hardware and software intersection point is MLIR. Therefore, MLIR-based CIRCT was selected as the hardware compilation infrastructure. Default parameterized DSA templates accompany the framework. Implementation options include Chisel, which is a high-level hardware programming language, or a DSA generator based on the Calyx infrastructure. Both Chisel and Calyx can access CIRCT in the compilation path, which can help unify the implementation of the co-design module. The DSA module can be connected to the buddy-codesign module for hardware–software co-design and to the buddy-mlir module for compilation and execution. In the future, after the completion of the aforementioned steps, the DSA module will be open source.

Co-design module

The co-design module is dedicated to the joint DSE of both software and hardware. Our strategy of configuring compiler passes and hardware designs simultaneously leads the co-design module to be decoupled as a separate infrastructure. The base components of the module include design space, search strategy, and cost model. Regarding these components, there are some mature studies on search strategies and cost models, and the appropriate strategy for the co-design module should be selected. Our innovation lies in the joint software and hardware search space at the IR level, which includes the hardware architecture, compilation optimization, and compilation pipeline. These design spaces are derived from the parameterized passes and the permutations of these passes in the software and hardware compilers. After DSE, specific parameter configurations can be used to guide code generation for both software and hardware. The search space increases geometrically because of the configuration of both software and hardware. Therefore, the difficulty of this module is pruning the search space to reduce search time. In the future, the co-design module will be open source together with the DSA module.

Compiler as a service module

The compiler-as-a-service (CAAS) module is the important entry point for the Buddy Compiler ecosystem. Various tools and dependencies in the entire framework need to be used together for co-design or end-to-end compilation, which is challenging to configure and handle. Therefore, our online platform integrates the entire toolchain to help users implement and demonstrate ideas quickly, so they can focus on their research without being blocked by the software and hardware environment. In particular, in the case of DSA emulators, using the buddy-caas can save considerable time in setting up the environment. Our user-friendly interface and configuration mechanism serve MLIR developers well for development and debugging. In addition, our platform integrates with the RISC-V toolchain, which has served many engineers and researchers without actual hardware; for example, our platform has powered MLIR abstraction design and integration tests on the RISC-V Vector extension. More generic features are in the development process to ease users into the project. In the future, we plan to integrate additional MLIR-related projects to build an entire online ecosystem.

Buddy Compiler contributions

The Buddy Compiler is designed to solve the problems described in the Current Problems and Future Directions section. As an open-source community, Buddy Compiler hopes to provide a compilation framework for hardware–software co-design, where researchers from different fields can contribute, promoting ecosystem development. This project makes the following contributions:

Buddy Compiler proposes a compiler framework for co-design. In our opinion, MLIR is currently the best unified abstraction method for both software and hardware. Both MLIR and MLIR-based CIRCT work as compiler infrastructures, and a framework is needed to integrate them to achieve a synergistic effect. In our opinion, hardware–software co-design is one of the killer applications of the MLIR ecosystem. Therefore, we propose a compiler framework based on MLIR to enable new co-design opportunities.

Buddy Compiler improves the efficiency of multi-level compilation. To improve the development efficiency of multi-level compilers and the execution efficiency of the generated code, we develop the buddy-caas online platform and an Auto-Config mechanism. Our buddy-caas platform can help developers quickly debug MLIR code and configure pass pipelines. Users can also easily conduct demonstration with the integrated backend environment. In addition, the purpose of the Auto-Config mechanism is to achieve hardware-aware optimization and code generation, which can improve execution efficiency and avoid duplicate development for multiple hardware platforms.

Buddy Compiler reduces the implementation overhead of domain-specific compilers. The implementation overhead for domain-specific compilers comes from frontend and backend support for applications and hardware. To address this issue, we built an Antlr-based frontend and RISC-V-oriented backend in buddy-mlir with customization and extensibility features. Thus, the infrastructure can be reused for the frontend parser, midend IR, and backend code generation. Domain-specific compiler implementation has become a declarative extension that substantially reduces engineering overhead.

Buddy Compiler builds a software and hardware compilation ecosystem. The purpose of building an ecosystem is to seamlessly integrate the features of various projects to reduce repeated engineering and foster synergy. Different MLIR-based projects may use different upstream versions, which leads to different interfaces and thus makes it difficult to compose a complete ecosystem. To solve this problem, we maintain a compilation ecosystem that can be integrated with other MLIR-based projects, such as a IREE and Torch-MLIR. In addition, we also have facilities to serve the compilation ecosystem, such as benchmark framework and a CAAS platform.

Summary

In the history of neural networks, software and hardware have promoted each other, and co-design has become increasingly important in the past decade. With the explosion of deep learning applications and hardware, there is a trend of fragmentation in deep learning systems, and compilation technologies are required to better bridge workloads to various hardware platforms and provide more opportunities for optimization. This paper reviews compilation technologies in deep learning co-design, thereby observing that compilation technologies are mainly applied to deep learning compilers and hardware compilation infrastructures. We believe that the current deep learning co-design is transiting from fragmentation to ecosystems, and there are still some problems that must be solved to form a unified ecosystem. This paper also summarized these problems along with future research directions. Based on this survey and analysis, we further propose a blueprint for a domain-specific compiler framework for co-design and share the current progress. In the golden age of computer architectures and domain-specific compiler, we believe that the co-design approach based on compilation technologies is the next breakthrough for future deep learning systems.

Acknowledgments

We would like to express our gratitude to Diego Caballero from Google, Hanchen Ye from the University of Illinois Urbana-Champaign, and Jiuyang Liu from Huazhong University of Science and Technology for their valuable comments and suggestions. We would like to thank the editorial board of Intelligent Computing, as well as Honggang Zhang and Jiayu Hua, for their helpful feedback and support. We would also like to thank Yuchen Li, Zhiyuan Tan, and Taiqi Zheng from the Institute of Software, Chinese Academy of Sciences for their assistance with this paper. **Funding**: This study was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant Nos. XDA0320000 and XDA0320200). **Author contributions:** H.Z. contributed to the compiler ecosystem design and paper writing. M.X. conducted co-design research of deep learning compilers. Y.W. composed the thematic structure and outline of this paper. C.Z. conducted background research on compiler technologies. All authors critically reviewed and approved the final version of the manuscript. **Competing interests:** The authors declare that they have no competing interests.

Data Availability

The data and materials used in this study are available upon request. Interested researchers can contact Y.W. at yanjun@iscas.ac.cn to request access to the data.

References

1. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Commun ACM*. 2017;60(6):84–90.
2. OpenAI, ChatGPT. [blog] Optimizing language models for dialogue. 2022 Nov 30. [accessed 27 March 2023] https://openai.com/blog/chatgpt/.
3. Deng L, Yu D. Deep learning: Methods and applications. *Found Trends Signal Process*. 2014;7(3-4):197–387.
4. Ramachandram D, Taylor GW. Deep multimodal learning: A survey on recent advances and trends. *IEEE Signal Process Mag*. 2017;34(6):96–108.
5. Sarker IH. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *SN Comput Sci*. 2021;2(6):Article 420.
6. Intel. Intel® architecture instruction set extensions and future features. 2021 May. [accessed 27 March 2023] https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf
7. Stephens N, Biles S, Boettcher M, Eapen J, Eyole M, Gabrielli G, Horsnell M, Magklis G, Martinez A, Premillieu N, et al. The arm scalable vector extension. *IEEE Micro*. 2017;37(2):26–39.
8. RISC-V Vector Extension Spec Contributors. RISC-V "V" Vector Extension, [accessed 27 March 2023] https://github.com/riscv/riscv-v-spec/releases/tag/v1.0
9. Markidis S, Chien SWD, Laure E, Peng IB, Vetter JS. Nvidia tensor core programmability, performance & precision. Paper presented at: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2018 Mar 11; Vancouver, BC, Canada.
10. Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, et al. In-datacenter performance analysis of a tensor processing unit. Paper presented at: Proceedings of the 44th Annual International Symposium on Computer Architecture; 2017 Jun 24–28; Toronto, Canada.

11. Liu S, Du Z, Tao J, Han D, Luo T, Xie Y, Chen Y, Chen T. Cambricon: An instruction set architecture for neural networks. *ACM SIGARCH Comput Archit News*. 2016;44(3):393–405.
12. Genc H, Kim S, Amid A, Haj-Ali A, Iyer V, Prakash P, Zhao J, Grubb D, Liew H, Mao H, et al., Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. Paper presented at: 2021 Dec 5–9 58th ACM/ IEEE Design Automation Conference (DAC); 2021 Dec 5–9; San Francisco, CA.
13. Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenk O. MLIR: Scaling compiler infrastructure for domain specific computation. Paper presented at: 2021 IEEE/ ACM International Symposium on Code Generation and Optimization (CGO); 2021 Feb 27–Mar 3; Korea.
14. Nigam R, Thomas S, Li Z, Sampson A. A compiler infrastructure for accelerator generators. Paper presented at: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems; 2021 Apr 19–23; USA.
15. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L. TVM: An automated end-to-end optimizing compiler for deep learning. Operating systems design and implementation. Paper presented at: OSDI'18: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation; 2018 Oct 8–10 Carlsbad, CA.
16. Google IREE Team. IREE (Intermediate Representation Execution Environment). [accessed 27 March 2023] https://github.com/openxla/iree
17. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Gan L, Yang G, Qian D. The deep learning compiler: A comprehensive survey. *IEEE Transac Parallel Distrib Syst*. 2020;32(3):708–727.
18. Xing Y, Weng J, Wang Y, Sui L, Shan Y, Wang Y. An in-depth comparison of compilers for deep neural networks on hardware. Paper presented at: 2019 IEEE International Conference on Embedded Software and Systems (ICESS); 2019 Jun 2–3; Las Vegas, NV.
19. Georganas E. Anatomy of high-performance deep learning convolutions on SIMD architectures. Paper presented at: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis; 2018 Nov 11–16; Denver, CO.
20. Mittal S, Vaishay S. A survey of techniques for optimizing deep learning on gpus. *J Syst Archit*. 2019;99:Article 101635.
21. TensorFlow XLA Contributors. XLA: Optimizing compiler for machine learning. [accessed 27 March 2023] https://www.tensorflow.org/xla
22. Vasilache N, Zinenko, Theodoridis T, Goyal P, DeVito Z, Moses WS, Verdoolaege S, Adams A, Cohen A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. ArXiv. 2018. https://doi.org/10.48550/arXiv.1802.04730
23. Nikhil R. Bluespec system verilog: Efficient, correct RTL from high level specifications. Paper presented at: Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04; 2004 Jun 23–25; San Diego, CA.
24. Bachrach J, Vo H, Richards B, Lee Y, Waterman A, Avižienis R, Wawrzynek J, Asanovic K. Chisel: Constructing hardware in a scala embedded language. Paper presented at: Proceedings of the 49th Annual Design Automation Conference; 2012 Jun 3–7; San Francisco, CA.
25. Koeplinger D, Feldman M, Prabhakar R, Zhang Y, Hadjis S, Fiszel R, Zhao T, Nardi L, Pedram A, Kozyrakis C. Spatial: A language and compiler for application accelerators. Paper presented at: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation; 2018 Jun 18–22; Philadelphia, PA.
26. Nigam R, Atapattu S, Thomas S, Li L, Bauer T, Ye Y, Koti A, Sampson A, Zhang Z, Predictable accelerator design with time-sensitive affine types. Paper presented at: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation; 2020 Jun 15–20; London, UK.
27. CIRCT Community. Circuit IR compilers and tools. [accessed 27 March 2023] https://circt.llvm.org/.
28. LeCun Y. 1.1 Deep learning hardware: Past, present, and future. Paper presented at: 2019 IEEE International Solid-State Circuits Conference (ISSCC); 2019 Feb 17–21; San Francisco, CA.
29. Dally WJ, Turakhia Y, Han S. Domain-specific hardware accelerators. *Commun ACM*. 2020;63(7):48–57.
30. Chen Y, Xie Y, Song L, Chen F, Tang T. A survey of accelerator architectures for deep neural networks. *Engineering*. 2020;6(3):264–274.
31. Peccerillo B, Mannino M, Mondelli A, Bartolini S. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *J Syst Archit*. 2022;Article 102561.
32. Reuther A, Michaleas P, Jones M, Gadepally V, Samsi S, Kepner J. AI and ML accelerator survey and trends. Paper presented at: IEEE High Performance Extreme Computing Conference (HPEC); 2022 Sep 19–23; Waltham, MA.
33. Teich J. Hardware/software codesign: The past, the present, and predicting the future. *Proc IEEE*. 2012;100(Special Centennial Issue):1411–1430.
34. Bringmann O, Ecker W, Feldner I, Frischknecht A, Gerum C, Hämäläinen T, Hanif MA, Klaiber MJ, Mueller-Gritschneder D, et al. Automated HW/SW co-design for edge AI: State, challenges and steps ahead. Paper presented at: Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis; 2021 Oct 10–13; New York, NY.
35. Hennessy JL, Patterson DA. A new golden age for computer architecture. *Commun ACM*. 2019;62(2):48–60.
36. Minsky M, Papert S. *Perceptron: An introduction to computational geometry*. Cambridge (MA): MIT Press; 1969.
37. Hopfield JJ. Neural networks and physical systems with emergent collective computational abilities. *Proc Natl Acad Sci U S A*. 1982;79(8):2554–2558.
38. Hinton G, Sejnowski T. Optimal perceptual inference. Paper presented at: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 1983 Jun 19; Washington, DC.
39. Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. *Nature*. 1986;323(6088):533–536.
40. Graf HP, Janow RH, Henderson D, Lee R. Reconfigurable neural net chip with 32k connections. Paper presented at: Advances in Neural Information Processing Systems 3; 1990 Oct 1; Denver, CO.
41. Boser BE, Sackinger E, Bromley J, Cun YL, Jackel LD. An analog neural network processor with programmable topology. *IEEE J Solid State Circuits*. 1991;26(12):2017–2025.

42. Cloutier J, Cosatto E, Pigeon S, Boyer FR, Simard PY. VIP: An FPGA-based processor for image processing and neural networks. Paper presented at: Proceedings of Fifth International Conference on Microelectronics for Neural Networks; 1996 Feb 12; Lausanne, Switzerland.
43. Collobert R, Bengio S, Mariéthoz J. Torch: A modular machine learning software library. Martigny (Switzerland): IDIAP; 2002.
44. Hill MD, Marty MR. Amdahl's law in the multicore era. *Computer*. 2008;41(7):33–38.
45. Hinton GE, Salakhutdinov R. Reducing the dimensionality of data with neural networks. *Science*. 2006;313(5786):504–507.
46. NVIDIA. CUDA toolkit. [accessed 27 March 2023]. https://developer.nvidia.com/cuda-toolkit
47. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. Paper presented at: MM '14: 2014 ACM Multimedia Conference; 2014 Nov 3–7; Orlando, FL.
48. Tokui S, Oono K, Hido S, Clayton J. Chainer: A next-generation open source framework for deep learning. Paper presented at: Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS); 2015 Dec 7–12; Montréal, Canada.
49. Al-Rfou R, Alain G, Almahairi A, Angermueller C, Bahdanau D, Ballas N, Bastien F, Bayer J, Belikov A, Belopolsky A, et al. Theano: A python framework for fast computation of mathematical expressions. ArXiv. 2016. https://doi.org/10.48550/arXiv.1605.02688
50. Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*. 2016;44(3):243–254.
51. NVIDIA. Nvidia tensor cores. [accessed 27 March 2023] https://www.nvidia.com/en-us/data-center/tensor-cores/.
52. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al. TensorFlow: A system for large-scale machine learning. Paper presented at: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16); 2016 Nov 2–4; Savannah, GA.
53. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. Pytorch: An imperative style, high-performance deep learning library. *Adv Neural Inf Proces Syst*. 2019;32:Article 721.
54. Seide F, Agarwal A. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. Paper presented at: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2016 Aug 13–17; New York, NY.
55. Chen T, Li M, Li Y, Lin M, WAng N, Wang M, Xiao T, Xu B, Zhang C, Zhang Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. ArXiv. 2015. https://doi.org/10.48550/arXiv.1512.01274
56. ONNX Community. ONNX: Open neural network exchange. [accessed 27 March 2023] https://onnx.ai/.
57. Mikolov T, Karafiát M, Burget L, Černocký J, Khudanpur S. Recurrent neural network based language model. Paper presented at: Proceedings of the 11th Annual Conference of the International Speech Communication Association (INTERSPEECH 2010); 2010 Sep 26–30; Chiba, Japan.
58. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention is all you need. Paper presented at: Advances in Neural Information Processing Systems 30 (NIPS 2017); 2017 Dec 4–9; Long Beach, CA.
59. Rotem N, Fix J, Abdulrasool S, Catron G, Deng D, Dzhabarov R, Gibson N, Hegeman J, Lele M, Levenstein R, et al. Glow: Graph lowering compiler techniques for neural networks. ArXiv. 2018. https://doi.org/10.48550/arXiv.1805.00907
60. Cyphers S, Bansal AK, Bhiwandiwalla A, Bobba J, Brookhart M, Chakraborty A, Constable W, Convey C, Cook L, Kanawi O, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. ArXiv. 2018. https://doi.org/10.48550/arXiv.1801.08058
61. De Michell G, Gupta RK. Hardware/software co-design. *Proc IEEE*. 1997;85(3):349–365.
62. Gupta RK, De, Micheli G. Hardware-software cosynthesis for digital systems. *IEEE Design Test Comput*. 1993;10(3):29–41.
63. Ernst R, Henkel J, Benner T. Hardware-software cosynthesis for microcontrollers. *IEEE Design Test Comput*. 1993;10(4):64–75.
64. Teich T, Blickle T, Thiele L. An evolutionary approach to system-level synthesis. Paper precented at: Proceedings of 5th International Workshop on Hardware/Software Co Design. Codes/CASHE'97; 1997 Mar 24–26; Braunschweig, Germany.
65. Blickle T, Teich J, Thiele L. System-level synthesis using evolutionary algorithms. *Des Autom Embed Syst*. 1998;3(1):23–58.
66. Lattner C. The golden age of compiler design in an era of HW/SW co-design. KEYNOTES AND INTERVIEWS in ASPLOS. [accessed 2021 Apr 22]. https://www.reddit.com/r/ProgrammingLanguages/comments/mvv24w/the_golden_age_of_compiler_design_in_an_era_of/
67. Iandola F, Han S, Moskewicz MW, Ashraf K, Dally WJ, Keutzer K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size: Computer Vision and Pattern Recognition. ArXiv. 2016. https://doi.org/10.48550/arXiv.1602.07360
68. Howard AG, Zhu M, Chen B, Kalenichenko D, Wang W, Weyand T, Andreett M, Adam H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. ArXiv. 2017. https://doi.org/10.48550/arXiv.1704.04861
69. Zhang X, Zhou X, Lin M, Sun J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. Paper presented at: Proceedings of the IEEE conference on computer vision and pattern recognition; 2018 Jun 18–22; Salt Lake City, UT.
70. Tung F, Mori G. CLIP-Q: Deep network compression learning by in-parallel pruning-quantization. Paper presented at: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; 2018 Jun 18–22; Salt Lake City, UT.
71. Han S, Mao H, Dally WJ. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. ArXiv. 2015. https://doi.org/10.48550/arXiv.1510.00149
72. Jacob B, Kligys S, Chen B, Zhu M, Tang M, Howard A, Adam H, Kalenichenko D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. ArXiv. 2018. https://doi.org/10.48550/arXiv.1712.05877
73. Jin Q, Yang Y, Liao Z. Towards efficient training for neural network quantization. ArXiv. 2019. https://doi.org/10.48550/arXiv.1912.10207
74. Wiedemann S, Shivapakash S, Becking D, Wiedemann P, Samek W, Gerfers F, Wiegand T. Fantastic4: A hardware-software co-design approach for efficiently running 4bit-

compact multilayer perceptrons. *IEEE Open J Circuits Syst*. 2021;2:407–419.
75. Chen W, Wang Y, Yang S, Liu C, Zhang L. You only search once: A fast automation framework for single-stage DNN/accelerator co-design. ArXiv. 2020. https://doi.org/10.48550/arXiv.2005.07075
76. Gupta S, Akin B. Accelerator-aware neural network design using AutoML. ArXiv. 2020. https://doi.org/10.48550/arXiv.2003.02838
77. Hao C, Zhang X, Li Y, Huang S, Xiong J, Rupnow K, W-m H, Chen D. FPGA/DNN co-design: An efficient design methodology for 1ot intelligence on the edge. Paper presented at: 2019 56th ACM/IEEE Design Automation Conference (DAC); 2019 Jun 2; Las Vegas, NV.
78. Jiang W, Zhang X, Sha EH-M, Yang L, Zhuge Q, Shi Y, Hu J. Accuracy vs Efficiency: Achieving both through FPGA-implementation aware neural architecture search. Paper presented at: 2019 56th ACM/IEEE Design Automation Conference (DAC); 2019 Jun 2–6; Las Vegas, NV.
79. Marculescu D, Stamoulis D, Cai E. Hardware-aware machine learning: Modeling and optimization. Paper presented at: IEEE/ACM International Conference on Computer-Aided Design; 2018 Nov 5; San Diego, CA.
80. Elsken T, Metzen JH, Hutter F. Neural architecture search: A survey. *J Mach Learn Res*. 2018;20(1):1997–2017.
81. Deng L, Li G, Han S, Shi L, Xie Y. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proc IEEE*. 2020;108(4):485–532.
82. Chen W, Wang Y, Xu Y, Gao C, Liu C, Zhang L. A framework for neural network architecture and compile co-optimization. *ACM Trans Embed Comput Syst*. 2022;22(1):1–24.
83. Lin J, Chen W-M, Lin Y, Cohn J, Gan C, Han S. MCUNet: Tiny deep learning on IoT devices. *Adv Neural Inf Proces Syst*. 2020;33:11711–11722.
84. Ma X, Guo F-M, Niu X, Lin X, Tang X, Ma K, Ren B, Wang Y. PCONV: The missing but desirable sparsity in DNN weight pruning for real-time execution on mobile devices. Paper presented at: Proceedings of the AAAI Conference on Artificial Intelligence; 2020 Feb 7; New York, NY.
85. Niu W, Ma X, Lin S, Wang S, Qian X, Lin X, Wang Y, Ren B. PatDNN: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning. Paper presented at: ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems; 2020 Mar 16–20; San Diego, CA.
86. Guan H, Liu S, Ma X, Niu W, Ren B, Shen X, Wang Y, Zhao P. CoCoPIE: Enabling real-time AI on off-the-shelf mobile devices via compression-compilation co-design. *Commun ACM*. 2021;64(6):62–68.
87. Krizhevsky A. One weird trick for parallelizing convolutional neural networks. ArXiv. 2014. https://doi.org/10.48550/arXiv.1404.5997
88. Song L, Mao J, Zhuo Y, Qian X, Li H, Chen Y. HyPar: Towards hybrid parallelism for deep learning accelerator array. Paper presented at: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA); 2019 Feb 16–20; Washington, DC.
89. Song L, Mao J, Zhuo Y, Qian X, Li H, Chen Y. AccPar: Tensor partitioning for heterogeneous deep learning accelerators. Paper presented at: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA); 2020 Feb 22–26; San Diego, CA.
90. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cuDNN: Efficient primitives for deep learning. ArXiv. 2014. https://doi.org/10.48550/arXiv.1410.0759
91. Intel oneAPI Deep Neural Network Library Team. Intel® oneAPI deep neural network library. [accessed 27 March 2023] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html
92. OpenBLAS Contributors. OpenBLAS: An optimized BLAS (Basic Linear Algebra Subprograms) library. [accessed 27 Mar 2023] https://github.com/xianyi/OpenBLAS
93. NVIDIA cuBLAS Team. cuBLAS. [accessed 27 March 2023] https://docs.nvidia.com/cuda/cublas/index.html
94. Hassan RO, Mostafa H. Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC. *Analog Integr Circ Sig Process*. 2021;106(2):399–408.
95. Ye H, Hao C, Cheng J, Jeong H, Huang J, Neuendorffer S, Chen D. ScaleHLS: A new scalable high-level synthesis framework on multi-level intermediate representation. Paper presented at: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA); 2022 Apr 2–6; Seoul, South Korea.
96. Thomas DE, Moorby PR. *The Verilog® hardware description language*. Berlin/Heidelberg (Germany): Springer Science & Business Media; 1990.
97. Feist T. Vivado design suite. *White Pap.* 2012;5:30.
98. Ajayi T, Blaauw D. OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain. Paper presented at: Proceedings of Government Microcircuit Applications and Critical Technology Conference; 2019 Mar 25; Albuquerque, NM.
99. Vissers K. Versal: The Xilinx Adaptive Compute Acceleration Platform (ACAP). Paper presented at: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays; 2019. Feb 24–26; Seaside, CA.
100. Kathail V. Xilinx Vitis unified software platform. Paper presented at: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays; 2020 Feb 23–25; Seaside, CA.
101. Venieris SI, Kouris A, Bouganis C-S. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Comput Surv*. 2018;51(3):56.
102. Nvidia NVDLA Team. NVDLA. [accessed 27 March 2023] http://nvdla.org/.
103. Chen T, Zheng L, Yan E, Jiang Z, Moreau T, Ceze L, Guestrin C, Krishnamurthy A. Learning to optimize tensor programs. *Adv Neural Inf Proces Syst*. 2018;31.
104. Zheng L, Jia C, Sun M, Wu Z, Yu C. H, Haj-Ali A, Wang Y, Yang J, Zhuo D, Sen K, et al. Ansor: Generating high-performance tensor programs for deep learning. Paper presented at: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation; 2020 Nov 4–6; Virtual Event.
105. Yang X, Gao M, Liu Q, Setter J, Pu J, Nayak A, Bell S, Cao K, Ha H, Raina P, et al. Interstellar: Using halide's scheduling language to analyze DNN accelerators. Paper presented at: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems; 2020 Mar 16–20; Lausanne, Switzerland.
106. Xi SL, Yao Y, Bhardwaj K, Whatmough PN, Wei G-Y, Brooks D. SMAUG: End-to-End full-stack simulation

infrastructure for deep learning workloads. ArXiv. 2019. https://doi.org/10.48550/arXiv.1912.04481
107. Wu YN, Emer JS, Sze V, Accelergy: An architecture-level energy estimation methodology for accelerator designs. Paper presented at: IEEE: Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD); 2019 Nov 4–7; Westminster, CO.
108. Parashar A, Raina P, Shao YS, Chen Y-H, Ying VA, Mukka A, Venkatesan R, Khailany B, Keckler SW, Emer J, Timeloop: A systematic approach to DNN accelerator evaluation. Paper presented at: IEEE: Proceedings of the 2019 International Symposium on Performance Analysis of Systems and Software; 2019 Mar 24–26; Madison, WI.
109. Dave S, Kim Y, Avancha S, Lee K, Shrivastava A. dMazeRunner: Executing perfectly nested loops on dataflow accelerators. *ACM Trans Embed Comput Syst*. 2019;18(5s):70.
110. Kwon H, Chatarasi P, Pellauer M, Parashar A, Sarkar V, Krishna T. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. Paper presented at: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture; 2019 Oct 12–16; Columbus, OH.
111. Venkatesan R, Shao YS, Wang M, Clemons J, Dai S, Fojtik M, Keller B, Klinefelter A, Pinckney N, Raina P, et al. MAGNet: A modular accelerator generator for neural networks. Paper presented at: IEEE: Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD); 2019 Nov 4–7; Westminster, CO.
112. Kirkpatrick S, Gelatt CD Jr, Vecchi MP. Optimization by simulated annealing. *Science*. 1983;220(4598):671–680.
113. Mitchell M. *An introduction to genetic algorithms*. Cambridge (MA): MIT Press; 1998.
114. Zoph B, Le QV. Neural architecture search with reinforcement learning. ArXiv. 2016. https://doi.org/10.48550/arXiv.1611.01578
115. Sohrabizadeh A, Yu CH, Gao M, Cong J. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transact Des Autom Electron Syst*. 2022;27(4):1–27.
116. Adams A, Ma K, Anderson L, Baghdadi R, Li T-M, Gharbi M, Steiner B, Johnson S, Fatahalian K, Durand F, et al. Learning to optimize halide with tree search and random programs. *ACM Trans Graph*. 2019;38(4):121.
117. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, techniques, & tools*. Noida, Uttar Pradesh (India): Pearson Education India; 2007.
118. Muchnick S. *Advanced compiler design implementation*. Burlington (MA): Morgan Kaufmann; 1997.
119. Appel AW. *Modern compiler implementation in C*. Cambridge (England): Cambridge Univ Press; 2004.
120. Tillet P, Kung H.-T, Cox D. Triton: An intermediate language and compiler for tiled neural network computations. Paper presented at: Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages; 2019 Jun 22; Phoenix, AZ.
121. Chen Y-H, Emer J, Sze V. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*. 2017;37(3):12–21.
122. Das A, Kumar A, Veeravalli B. Energy-aware task mapping and scheduling for reliable embedded computing systems. *ACM Trans Embed Comput Syst*. 2014;13(2s):72.
123. Zhao J, Li B, Nie W, Geng Z, Zhang R, Gao X, Cheng B, Wu C, Cheng Y, Li Z, et al., AKG: Automatic kernel generation for neural processing units using polyhedral transformations. Paper presented at: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation; 2021 Jun 20–25; Virtual Event, Canada.
124. Lai Y-H, Chi Y, Hu Y, Wang J, Yu CH, Zhou Y, Cong J, Zhang Z, HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. Paper presented at: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays; 2019 Feb 24–26; Seaside, CA.
125. Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. Paper presented at: IEEE: Proceedings of the International Symposium on Code Generation and Optimization, 2004 CGO; 2004 Mar 20–24; San Jose, CA.
126. TensorFlow Community. MLIR-HLO: A standalone HLO MLIR-based compiler. [accessed 27 March 2023] https://github.com/tensorflow/mlir-hlo
127. Torch-MLIR Community. The Torch-MLIR project. [accessed 27 March 2023] https://github.com/llvm/torch-mlir
128. ONNX Community. ONNX-MLIR. [accessed 27 March 2023] http://onnx.ai/onnx-mlir/
129. Roesch J, Lyubomirsky S, Weber L, Pollock J, Kirisame M, Chen T, Tatlock Z. Relay: A new IR for machine learning frameworks. Paper presented at: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages; 2018 Jun 18; Philadelphia, PA.
130. Fehr M, Niu J, Amini R. Riddle,M, Su Z, Grosser T. IRDL: An IR definition language for SSA compilers. Paper presented at: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation; 2022 Jun 13–17; San Diego, CA.
131. TVM Community. Introduction to Relay IR. [accessed 27 March 2023] https://tvm.apache.org/docs/arch/relay_intro.html
132. CIRCT Community. CIRCT charter. [accessed 27 March 2023] https://circt.llvm.org/docs/Charter/
133. Guo K, Zeng S, Yu J, Wang Y, Yang H. A survey of FPGA-based neural network inference accelerators. *ACM Trans Reconfigurable Technol Syst*. 2019;12(1):2.
134. Intel. Intel® high level synthesis compiler. [accessed 27 March 2023] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html
135. Siemens EDA. High-level synthesis & verification. [accessed 27 March 2023] https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/.
136. Xilinx. Vivado design suite user guide: High-level synthesis (UG902) [accessed 27 March 2023] https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis
137. Cong J, Fan Y, Han G, Jiang W, Zhang Z. Platform-based behavior-level and system level synthesis. Paper presented at: IEEE: Proceedings of the 2006 IEEE International SOC Conference; 2006 Sep 24–27; Austin, TX.
138. Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Anderson JH, Brown S, Czajkowski T, LegUp: High-level synthesis for FPGA-based processor/accelerator systems. Paper presented at: Proceedings of the 19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays; 2011 Feb 27; Monterey, CA.
139. Gupta S, Gupta R, Dutt ND, Nicolau A. *SPARK: A parallelizing approach to the high-level synthesis of digital circuits*. Berlin/Heidelberg (Germany): Springer Science & Business Media; 2007.

140. Cong J, Zhang Z. An efficient and versatile scheduling algorithm based on SDC for mulation. Paper presented at: IEEE: Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference; 2006 Jul 24–28; San Francisco, CA.
141. Xilinx. SDSoC profiling and optimization guide. [accessed 27 March 2023] https://www.xilinx.com/support/documents/sw_manuals/xilinx2019_1/ug1235-sdsoc-optimization-guide.pdf
142. Intel. Introduction to Intel® FPGA SDK for OpenCL™ pro edition best practices guide. [accessed 27 March 2023] https://www.intel.com/content/www/us/en/docs/programmable/683521/22-3/introduction-to-pro-edition-best-practices.html
143. Schwartz J, Sharma NN, Rad D, Takusagawa K, Stoy J, Nikhil RS. The open-source Bluespec BSC compiler and reusable example designs. Paper presented at: Workshop on Open-Source EDA Technology (WOSET); 2021 Nov 4; Munich, Germany.
144. Izraelevitz A, Koenig J, Li P, Lin R, Wang A, Magyar A, Kim D, Schmidt C, Markley C, Lawson J, et al. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. Paper presented at: IEEE: Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICAD); 2017 Nov 13–16; Irvine, CA.
145. Asanovic K, Avizienis R, Bachrach J, Beamer S, Biancolin D, Celio C, Cook H, Dabbelt D, Hauser J, Izraelevitz A, et al. The rocket chip generator. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-17 (2016).
146. Prabhakar R, Zhang Y, Koeplinger D, Feldman M, Zhao T, Hadjis S, Pedram A, Kozyrakis C, Olukotun K. Plasticine: A reconfigurable architecture for parallel patterns. Paper presented at: Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA); 2017 Jun 24–28; Toronto, ON, Canada.
147. Dario A, Danny H. AI and compute [accessed 27 March 2023] https://openai.com/research/ai-and-compute
148. Zhu K, Zhao W, Zheng Z, Guo T, Zhao P, Bai J, Yang J, Liu X, Diao L, Lin W. DISC: A dynamic shape compiler for machine learning workloads. Paper presented at: Proceedings of the 1st Workshop on Machine Learning and Systems; 2021 Apr 26, Edinburgh, Scotland, UK.
149. Bik A, Koanantakool P, Shpeisman T, Vasilache N, Zheng B, Kjolstad F. Compiler support for sparse tensor computations in MLIR. *ACM Trans Archit Code Optim*. 2022;19(4):50.
150. Tian R, Guo L, Li J, Ren B, Kestor G. A high performance sparse tensor algebra compiler in MLIR. Paper presented at: IEEE: Proceedings of the 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC); 2021 Nov 14; St. Louis, MO.
151. Hu P, Lu M, Wang L, Jiang G. TPU-MLIR: A compiler for TPU using MLIR. ArXiv. 2022. https://doi.org/10.48550/arXiv.2210.15016
152. Bondhugula U. High performance code generation in MLIR: An early case study with GEMM. ArXiv. 2020. https://doi.org/10.48550/arXiv.2003.00532
153. Katel N, Khandelwal V, Bondhugula U. High performance GPU code generation for matrix-matrix multiplication using MLIR: Some early results. ArXiv. 2021. https://doi.org/10.48550/arXiv.2108.13191