An Automated Customizable Framework for Neural Network Acceleration on FPGAs

Zhuangwen Yang

Abstract—**Neural networks have been widely applied in industrial production and daily life. However, the constraints of the production environment make it difficult to deploy neural networks to the unified devices. In this paper, we present a neural network accelerator generator that can generate different accelerators based on different parameters. Different from the existing mainstream neural network accelerators, we utilize the novel SpinalHDL language to design the generator to achieve the balance between performance and flexibility. To facilitate the deployment of neural networks in the production environment, we propose a comprehensive toolchain including a TVM-based compiler and a SystemC-based simulator. The compiler optimizes the network operators and generates configuration instructions. The simulator is employed for algorithm modeling and system simulation, which establishes a versatile framework for neural network acceleration. We demonstrate the effectiveness of our approach by testing it on chips such as XCVU9P-2FSGD2104E using neural networks such as YOLOv4-Tiny and YOLOX. We perform accelerator validation on both single-core and multi-core architectures, which demonstrated favorable acceleration performance.**

Index Terms—**A framework for neural network acceleration, Hardware-software co-design, Hardware accelerator generator, Field-programmable gate array (FPGA)**

I. Introduction

Data, algorithms and computing power are the three keys that are behind the advancement of deep learning. In the recent years, deep learning has achieved remarkable success in various fields. However, the superior performance of deep learning faces to the expense of significant computational resources. Nevertheless, the growth rate of computational resources in chips is far away from the computational demands of algorithms. Although many neural network accelerators have been developed[1]-[12], their speed is also far behind the requirement of neural network algorithms. As a result, many researchers made efforts to construct a comprehensive suite of design tools for neural network accelerators. To expedite accelerator development, an agile development process is employed in accelerator design. Currently, FPGA-based neural network accelerators are often designed to accelerate specific neural network models and necessitate a redesign of the accelerator when a new model needs to be accelerated. Additionally, the absence of a comprehensive toolchain poses a significant challenge to the deployment of neural network accelerators.

This paper proposes a neural network accelerator generator based on SpinalHDL[13]. SpinalHDL enables the agile design of

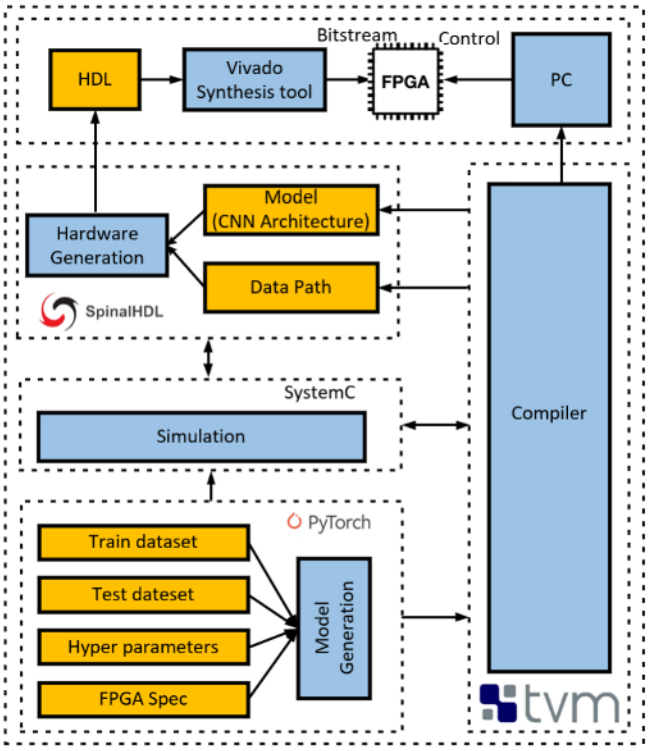

Figure 1 The overall framework for neural network acceleration

a generic neural network accelerator generator. By selecting different configuration parameters during Verilog generation, a specialized neural network accelerator can be produced to meet different requirements. The advancement of neural network algorithms has been remarkably swift, which enables the efficient deployment of newly released networks onto existing neural network generators. This paper proposes a TVM-based[14] compiler and a SystemC-based simulator for the deployment of neural network algorithms. The compiler optimizes the network operators and generates configuration instructions. The simulator is employed for algorithm modeling and system simulation. The toolchain facilitates the agile implementation of neural network algorithms as it is illustrated in Figure 1.

The major contributions of our work are summarized as the following:

(1) This paper proposes a comprehensive toolchain comprising a SpinalHDL-based generator, a TVM-based compiler and a SystemC-based simulator. These establishes an automated customizable framework for neural network acceleration.

(2) The application of agile development and

Manuscript received April 16, 2024

Zhuangwen Yang, School of computer science and technology, Tiangong University, Tianjin, China.

hardware-software co-design has resulted in a balanced performance-flexibility trade-off for the accelerator.

II. BACKGROUND AND RELATED WORK

A. Agile Design

The expeditious pace of development in neural network algorithms demands the neural network accelerators to facilitate swift iteration and development. FPGA is a well-suited acceleration platform in this regard. Currently, three principal methodologies exist in FPGA development. The first one involves employing hardware description languages such as Verilog and VHDL. The second one involves utilizing high-level synthesis languages, such as AMD XILINX's HLS. The third one employs domain-specific language tools specifically designed for the field, such as SpinalHDL and Chisel[15]. While existing HDL languages such as Verilog achieve widespread adoption, SpinalHDL outperforms them with advanced error detection capabilities and shorter development cycles, making SpinalHDL more conducive to agile design. Another approach generates the accelerator automatically based on pre-written code using template matching[16]-[18]; however, compared to SpinalHDL, it exhibits diminished flexibility. The adoption of HLS accelerates development cannot precisely control area and timing, resulting in suboptimal quality of generated RTL code that fails to meet performance requirements.

SpinalHDL is a novel hardware description language based on the extension of Scala. Compared to Verilog, it supports a broader range of object-oriented programming features and more advanced parameterization. Consequently, it offers more convenient and efficient capabilities and can adapt to the rapid iteration of neural networks. Currently, SpinalHDL supports simulation with various tools for simulation such as Verilator, VCS, and Vivado's XSIM. In summary, SpinalHDL has strong engineering development capabilities, making it highly suitable for agile design of neural network accelerators.

B. Model Quantization

Quantization involves converting the original floating-point model into alternative data formats for storage. There are currently two types of methods for compressing deep learning models. The first one involves constructing a lightweight network model, such as MobileNet[19]. The second one involves compressing the model through operations such as quantization and pruning. Quantizing models can reduce the data bit-width, thereby reducing computational complexity. For instance, quantizing 32-bit floating-point data to 8-bit integers can significantly reduce energy consumption according to Horowitz[20]. However, reducing the data bit-width too much results in accuracy issues with the model. Google's 2018 white paper on quantization showed that 8-bit quantization brings almost no loss in accuracy[21]. Therefore, we adopt 8-bit quantization in this paper.

III. A FRAMEWORK FOR NEURAL NETWORK ACCELERATION

A. Hardware-Software Co-Design

The overall framework for neural network acceleration proposed in this paper is illustrated in Figure 1. This framework includes five fundamental components: algorithm training, simulator design, compiler design, generator design and on-board deployment. In the first step, a neural network is trained, and the trained weight data is obtained. In the next step, the design space is explored using the simulator to identify the optimal configurations. The third step involves generating the neural network accelerator using the generator. In the fourth step, the compiler extracts the computation graph of the neural network and generates instructions. Finally, the design is deployed to an FPGA board. It should be noted that the exploration of the design space, generation of the accelerator and extraction of the computation graph by the compiler are not sequential processes. These three components require data interaction, such as the compiler providing the computation graph to the simulator. If the simulator identifies that the neural network needs to be segmented, the compiler must re-segment the neural network and construct a new computation graph. The new computation graph is then provided to the simulator for a new round of simulation.

Numerous studies have been conducted on the hardware-software co-design in the neural network acceleration field. For instance, Ahmed Nasser et al[22]. proposed a convolutional neural network accelerator framework that generates neural network operators via Perl scripts and creates the corresponding accelerator by varying neural network parameters. Nonetheless, this approach suffers from limited flexibility. In contrast, the framework proposed in this paper provides higher flexibility. The framework can be divided into two parts: software and hardware. The software portion consists of three components: a compiler, simulator, and trainer. The hardware portion is a generator. In the proposed neural network acceleration framework, a single language was not chosen for development. Instead, Python and SystemC are utilized for the software portion, while SpinalHDL and Verilog are employed for hardware description. Information exchange between the software and hardware portions was implemented in the JSON format. This approach was selected to maximize the performance advantages provided by each individual component.

When conducting hardware-software co-design, one of the primary concerns is ensuring coordination between different modules. When it comes to information exchange between the compiler and simulator, the neural network model is the primary piece of data that needs to be transmitted. This can be achieved through the use of a trained PTH file, with only the quantized model needing to be transferred to the compiler. To explore the design space, the compiler and simulator need to be coordinated. The simulator requires information such as the neural network's parameters, weights, and initial FPGA design architecture. The compiler also needs the same information. The compiler obtains the weights of the current neural network from the trainer and then optimizes them before passing them to the simulator. Once the design space exploration is complete, the FPGA design architecture is determined. After completing the design space exploration, the FPGA design architecture is determined. Once all parameters are finalized, the hardware accelerator can be generated. The generator, simulator, and compiler exchange data in JSON format. The generator generates corresponding RTL code and TCL files for synthesis, layout, and other operations in Vivado. After generating the bitstream, the weight and instruction files generated by the compiler need to be read on the host computer.

Table I
Configurable parameters of the generator

Param Type	Parameters	Values				Default Value
General parameters	data bit width	4	8	16	32	4
	image type	RGB	Gray			Gray
	convolution type	320	416	640		640
	operator type	1x1(8x)	1x1(4x)	1x1(2x)	1x1	1x1(8x)
	operator selection	Focus enable				no
	activation function type	Leaky Relu	Relu			Leaky Relu
	on-chip cache size(MiB)	1	2	4		1
	off-chip storage size(GiB)	1	2	4		4
Performance parameters	computational parallelism	4x4	8x8	8x16	16x16	8x8
	maximum parallelism	16x16	32x32			16x16
	number of computing cores	1	2	3	4	1
	burst transfer size(B)	16	32	64		32
	dedicated IP enable	yes	no			yes
Optimization parameters	DSP frequency multiplication	1x	2x			1x
	DSP multiplexing	1x	2x			2x
	URAM enable	yes	no			yes
	delay of multipliers	3	4	5	6	3
	delay of adders	1	2	3		1
	delay of cache units	1	2	3	4	2
	timing optimization enable	yes	no			yes

B. Generator Design

The neural network generator proposed in this paper incorporates a series of configuration statements, which facilitate the creation of different accelerator configurations as it is shown in Table I. This approach minimizes resource consumption as the generator avoids producing redundant operators that do not require acceleration. The choice of FPGAs as the hardware platform was motivated by their

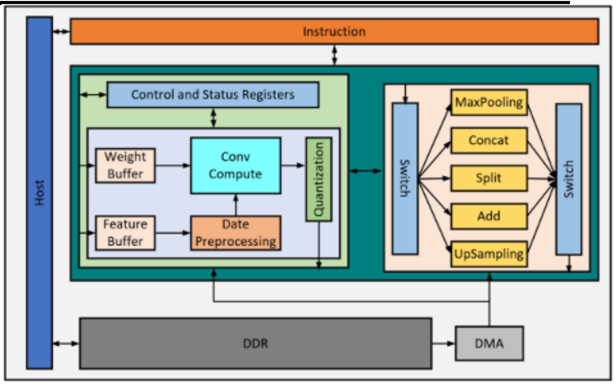

Fig. 2. Generator design

reconfigurable nature. Apart from accelerating various operators based on the neural network model, the proposed accelerator can generate distinct neural network accelerator architectures automatically that depends on the available FPGA chip resources. In addition, Table Ⅱ provides a comprehensive description of how the instruction register set controls the computational process, facilitates the transfer of network parameters, and enables the transmission of data.

Figure 2 illustrates the hardware acceleration diagram of the convolutional neural network accelerator proposed in this paper. It comprises three modules: the Conv module, responsible for convolution computation; the Instruction module, which controls functionality; the Shape module, which modifies the feature map dimensions. Both the Conv module and the Shape module can receive data from external storage modules and exchange data with each other, reducing the need for external memory access, improving computational speed, and minimizing power consumption.

Table Ⅱ
Instruction Register Set

Instruction Type	Register		Address	Field Description
Control	Conv	StateReg	[31:0]	State of the convolution
Computation		ControlReg	[31:0]	Control of the convolution
	Shape	StateReg	[31:0]	State of the shape
		ControlReg	[31:0]	Control of the shape
Network	Conv	ImageSizeReg	[31:22]	Number of input channels
Parameters			[21:11]	Number of input columns
			[10:0]	Number of input rows
		ParamReg	[31]	Stride enable
			[30:23]	Z3
			[22:20]	Number of Z1
			[19:12]	Z1
			[11]	Activation enable
			[10]	Padding enable
			[9:0]	Number of output channels
		ConvTypeReg	[31:16]	First layer
			[15:0]	Convolution type
		ParamCountReg	[31:16]	Number of quantified params
			[15:0]	Number of weight params
		AmendmentReg	[31:0]	Amendment
	Shape	DataSizeReg	[31:22]	Number of input channel 1
			[21:11]	Number of input columns
			[10:0]	Number of input rows
		C2Reg	[31:0]	Number of input channel 2
		S1Reg	[31:0]	Scale 1
		S2Reg	[31:0]	Scale 2
		Z1Reg	[31:0]	Zero 1
		Z2Reg	[31:0]	Zero 2
Data Transfer	Conv	WAddrReg	[31:0]	DMA write address

	WLenReg	[31:0]	DMA write length
	RAddrReg	[31:0]	DMA read address
	RLenReg	[31:0]	DMA read length
Shape	WAddrReg	[31:0]	DMA write address
	WLenReg	[31:0]	DMA write length
	RAddrReg	[31:0]	DMA read address
	RLenReg	[31:0]	DMA read length

In the field of neural networks, convolutional layers constitute the majority of the computational workload. Thus, optimizing the performance of these layers is critical. Conversely, operations that alter the feature map dimensions, such as pooling, upsampling, concatenation, addition and splitting occur infrequently. To efficiently manage these operations, they are grouped together into a module named Shape. The Instruction module is responsible for selecting and switching between different computation modes within the Conv module and Shape module.

The convolution acceleration module can be partitioned into several sub-modules, including the cache module for weight and feature maps, the convolution computation module and the quantization module, as it is illustrated in Figure 2. In particular, the convolution computation module employs the reuse and clock frequency doubling of DSP units to improve the accelerator's performance.

AMD XILINX's FPGA chips provide a range of DSP units such as DSP48 and DSP58. DSP48E1 and DSP48E2 can perform multiplications of 25 × 18 and 27 × 18 bits, respectively, while the DSP58 can perform multiplications of 27×24 bits. Since 8-bit quantization technology is employed in this paper, using a DSP unit solely with an 8-bit multiplication would result in a significant waste of DSP resources. Therefore, we employ DSP multiplexing technology[23]. Taking DSP48E1 as an example. DSP48E1 offers a calculation mode of (A+D) × B, which enables the conversion of A×B and D× B to (A+D) × B. Consequently, two 8-bit multiplications can be computed directly through this single DSP unit. Using DSP multiplexing technology can reduce DSP usage by half with the same level of parallelism configuration, which means that the computing power can be doubled within the same FPGA chip.

The DSP computation unit undertakes the execution of multiplication and accumulation operations within convolution processes, occupying a crucial position along the critical path of convolution calculations. When employing an FPGA as a carrier for neural network acceleration, the operating frequency is typically set to 200MHz or lower, which limits the DSP frequency to only 15 % to 30 % of the maximum frequency, thereby wasting DSP resources[24]. This is because leakage power is independent of clock frequency. As a result, the power consumption of the DSP is mostly wasted. Therefore, the computing power can be further enhanced by setting the DSP frequency to twice the clock frequency.

The Shape module includes functional modules such as MaxPooling, UpSampling and Concat. These modules reuse the same set of external data, which is distributed to the corresponding working modules via the Switch module after entering the Shape module. Once the computation is completed, the Switch module collects the results and sends them to the external storage unit. The functional modules can be pruned based on the requirements of the neural network algorithm, thereby they enable the resulting neural network accelerator to achieve resource minimization.

To increase the computational performance in such architectures, parallelism of the convolutional unit is often augmented. However, this brute-force parallelism augmentation is commonly limited by bandwidth, causing suboptimal acceleration results. Therefore, this paper proposes a multi-core architecture design. The design methodology of integrating multiple processing units on a single processor has been widely employed in CPU design and has been implemented in the proposed neural network accelerator.

This paper presents an analysis of the data flow in a multi-core architecture accelerator, using SqueezeNet as a case study. SqueezeNet mainly consists of Fire layers as it is illustrated in Figure 3(a). In the Fire layer, two convolutional layers share the same input. In a typical neural network accelerator, one convolutional layer is computed firstly, which is followed by the other and leads to redundant reads by the same input data from external storage and resulting in time and power wastage. To address this issue, our proposed multi-core architecture connects both input channels of Conv2 and Conv3 to the output channel of Conv1, as shown in Figure 3(a). This optimization allows the simultaneous calculation of both convolutional layers, resulting in a significant improvement in computational efficiency. The multi-core acceleration architecture diagrams of YOLOv4 and YOLOX are shown in Figure 3(b) and Figure 3(c), respectively.

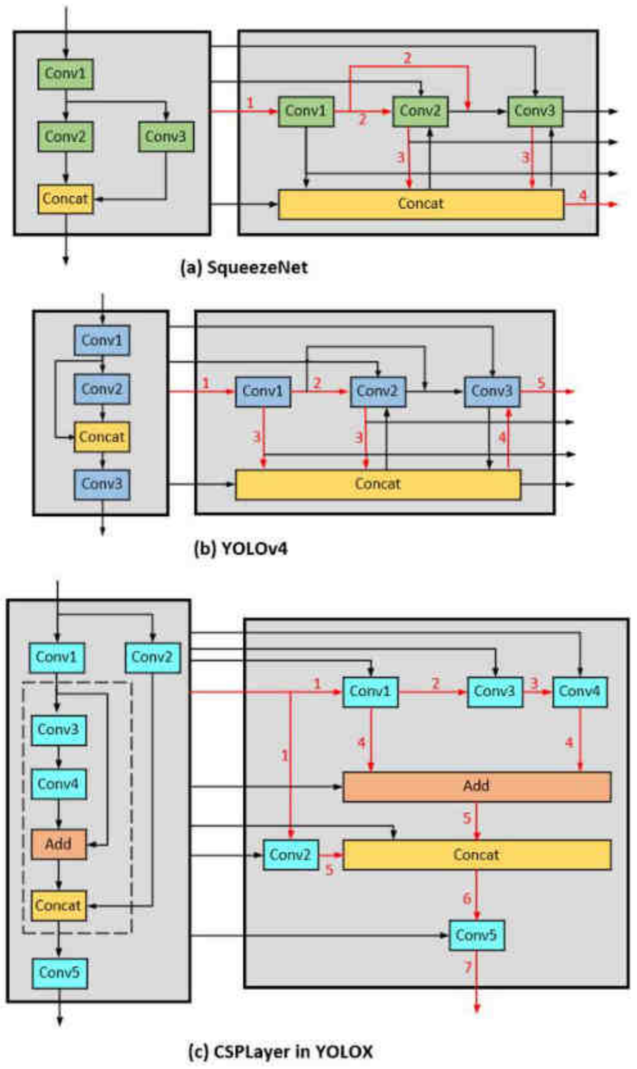

Fig. 3. Multi-core acceleration architecture

C. Toolchain Design

A well-designed toolchain can facilitate the transformation of a neural network accelerator from a mere functional unit to an efficient computing system. In this paper, we present a compiler and simulator designed for the deployment of neural network models. The compiler is an essential component of neural network acceleration. The compiler implemented in this paper focuses on quantization, optimization, and mapping of the model.

TVM is an end-to-end compilation and optimization stack that enables the deployment of neural network models generated by PyTorch, TensorFlow, and other frameworks onto various hardware platforms. In this paper, we extend the compiler based on TVM. The extended compiler efficiently map the neural network model onto the hardware architecture. In order to better support custom hardware, it is sometimes necessary to define custom operations. Therefore, TVM supports the use of custom passes to achieve this goal. Depending on the amount of FPGA resources available, a convolution operator can be split into multiple convolutions and concatenation operations, enabling large neural networks to be deployed on smaller FPGAs. The acceleration capabilities of the accelerator platform can be fully utilized. As it is shown in Figure 4, a convolution operator is split into four convolutions and three concatenation operations. This paper presents a compiler that extracts parameters, optimizes computation graphs and schedules instructions for deployment. By utilizing this compiler, neural networks can be automatically deployed while concealing hardware-specific parameters from software engineers, streamlining the process of neural network deployment.

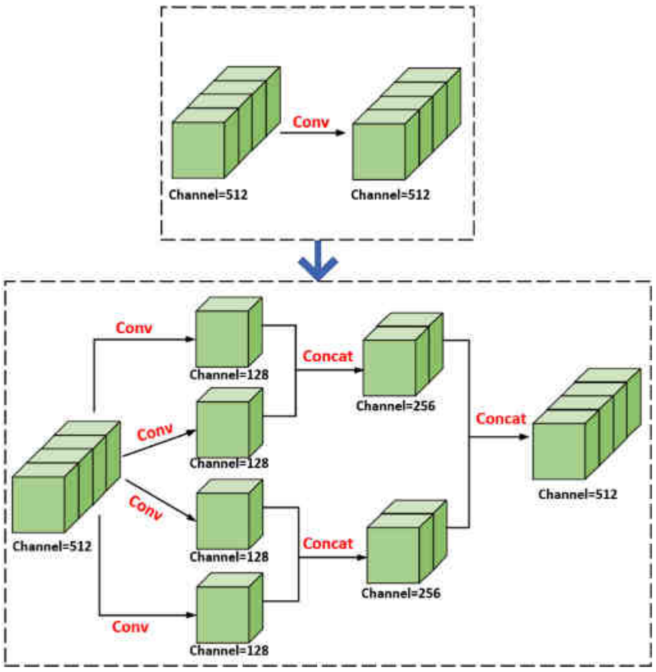

Fig. 4. A customized pass

The conventional approach to chip development often separates software and hardware development, leading to limited collaboration between software and hardware engineers. To solve this problem, SystemC was developed as a C++ library that enables software developers to rapidly construct hardware designs. SystemC extends C++ by providing functions and classes for hardware modeling with a simulation kernel that allows modeling from the Register Transfer Level to the system level. By utilizing SystemC, engineers can perform system-level modeling, conduct swift simulations and verifications, explore and evaluate various design architectures during the early development stages, and compare functional simulations with RTL descriptions during the later stages of development. Thus, SystemC promotes collaborative design between software and hardware engineers. In this paper, we utilize the SystemC simulator to explore the design space that encompasses power analysis and resource utilization across different architectures. Early adoption of SystemC for simulation modeling can alleviate the pressure of using RTL design for accelerator implementation in the later stages. Moreover, it is essential to maintain consistency between the simulation model and the RTL implementation. It enables exploration of new design iterations on the simulation model and delaying RTL design until the design has matured, which results in accelerated development speed and reduced costs.

IV. Experimental Results

To demonstrate the superior performance and ease of deployment of the proposed accelerator generator on different FPGA platforms, we conducted a series of experiments on a range of FPGA platforms, including AMD XILINX's XCVU9P-2FSGD2104E, XC7K325T-FFG900I, and XC7Z100-2FFG900I. YOLOv4-Tiny and YOLOX-S were selected for evaluation using the VOC2007 dataset, which were trained and quantized by using the PyTorch 1.7.

Table Ⅲ presents a comparison of the acceleration performance of our work with existing works. Considering multiple aspects such as speed, accuracy, and power consumption. Our work exhibits an attractive performance. Despite the lower power consumption of Jetson Nano compared to our design, our achieved frames per second (FPS) significantly surpass it. Consequently, when evaluating the ratio of FPS to power consumption, our design still demonstrates superior performance. Table Ⅳ and Ⅴ present comprehensive data on resource utilization, power consumption, and other relevant metrics for both single-core and multi-core architectures, respectively. These tables demonstrate the experimental results of the proposed neural network accelerator generator in this paper under single-core, multi-core architectures, and different clock frequencies, which comprehensively show the superior performances compared with other ones.

Table Ⅲ
Performance comparison of YOLOv4-Tiny acceleration on various platforms

	R7-5800H	RTX 3070	Jetson Nano	XCKU040[25]	VU9P[26]	VU9P(Our)
Platform	CPU	GPU	GPU	FPGA	FPGA	FPGA
Clock Frequency(GHz)	3.2	1.5	0.922	0.143	0.2	0.2
Data Type	FP32	FP32	FP32	FP16	INT8	INT8
FPS	19.53	131.98	14.26	31.20	49.38	102.75
mAP	78.60%	78.60%	78.60%	/	81.54%	76.40%
Power(W)	45	220	10	/	12.689	14.360

Table Ⅳ
Resource Utilization and Acceleration Performance of Single-Core Architecture under different parallelism, FPGA chips, and clock frequencies

		325T	7100	VU9P		VU9P(dsp2x)	VU9P(Our)	
	Compute Channel In	8	8	4	8	8	16	16
	Compute Channel Out	8	8	4	8	8	8	16
Resource	LUT	66920	41201	58419	74914	102071	94511	133095
	LUT RAM	10107	6055	6867	9565	9709	13095	19173
	FF	101206	70700	81638	112760	224601	153644	227846
	BRAM	266.5	243	120	164	164	232.5	244.5
	URAM	0	0	37	73	73	145	290
	DSP	474	474	173	477	333	885	1525
	NPU Power(W)	4.527	4.478	0.761	1.953	4.835	3.521	6.8
	Total Power(W)	10.768	8.176	7.802	9.015	11.971	10.636	13.958
YOLOv4-Tiny	FPS	22.22	24.06	6.56	23.04	/	45.81	87.58

	Performance(GOP/s)	152.52	165.15	45.03	158.15	/	314.44	601.15
	Energy Efficiency(GOPS/W)	14.164	20.199	5.772	17.543	/	29.564	43.068
YOLOX-S	FPS	15.09	16.07	3.94	15.14	/	29.07	54.92
	Performance(GOP/s)	172.20	183.39	44.96	172.78	/	331.78	626.69
	Energy Efficiency(GOPS/W)	15.992	22.430	5.763	19.166	/	31.194	44.898

Table V
Resource utilization and acceleration performance of multi-core architecture under different numbers of cores and FPGA chips

		7100		VU9P			
	Core Number	1	2	1	2	3	4
Resource	LUT	41201	65475	74914	101980	128408	155931
	LUT RAM	6044	9700	9565	13284	16999	20712
	FF	70700	120415	112760	159059	205612	251687
	BRAM	243	459.5	164	229	294	359
	URAM	0	0	73	146	219	292
	DSP	474	827	477	830	1183	1536
	NPU Power(W)	4.478	8.338	1.953	3.576	5.317	7.042
	Total Power(W)	8.176	12.167	9.015	10.700	12.572	14.360
YOLOv4-Tiny	FPS	24.06	55.82	23.04	53.91	78.80	102.75
	Performance(GOP/s)	165.15	383.15	158.15	370.04	540.88	705.28
	Energy Efficiency(GOPS/W)	20.199	31.491	17.543	34.583	43.02	49.11
YOLOX-S	FPS	16.07	37.12	15.14	35.28	49.81	66.92
	Performance(GOP/s)	183.39	423.61	172.78	402.62	569.57	763.69
	Energy Efficiency(GOPS/W)	22.430	34.816	19.166	37.628	45.305	53.182

V. Conclusion

In this paper, a high-performance neural network accelerator generator is presented, which is implemented by SpinalHDL. This generator can adapt to various FPGA chips, generating specialized accelerators optimized for energy efficiency or universal neural network accelerators capable of supporting multiple neural network algorithms. Additionally, a comprehensive toolchain is proposed. The compiler extracts the computation graph and neural network parameters, splits the neural network model and generates instructions to control accelerator operations. The simulator is responsible for functional simulation, design space exploration, resource and power estimation. The generator, compiler, and simulator collectively constitute a neural network acceleration framework. Through hardware-software co-design and agile design, our proposed neural network accelerator can quickly and easily deploy different neural networks to various FPGA chips. Finally, we conducted experiments to implement YOLOv4-Tiny and YOLOX-S on the 325T, 7100, and VU9P chips. We performed accelerator validation on both single-core and multi-core architectures, which demonstrates the better acceleration performance compared with other ones.

REFERENCES

[1] Islam M N, Shrestha R, Chowdhury S R, “A New Hardware-Efficient VLSI-Architecture of GoogLeNet CNN-Model Based Hardware Accelerator for Edge Computing Applications," 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE, 2022: 414-417.

[2] Zhang J, Yang T, Li Q, et al, “An FPGA-Based Neural Network Overlay for ADAS Supporting Multi-Model and Multi-Mode," 2021 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2021: 1-5.

[3] Liu X, Yang J, Zou C, et al. “Collaborative edge computing with FPGA-based CNN accelerators for energy-efficient and time-aware face tracking system," IEEE Transactions on Computational Social Systems, 2021, 9(1): 252-266.

[4] Zacchigna F G, ``Methodology for CNN Implementation in FPGA-based Embedded Systems," IEEE Embedded Systems Letters, 2022.

[5] Neris R, Rodríguez A, Guerra R, et al, “FPGA-Based Implementation of a CNN Architecture for the On-Board Processing of Very High-Resolution Remote Sensing Images," IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2022, 15: 3740-3750.

[6] Zhang G, Zhao K, Wu B, et al, “A RISC-V based hardware accelerator designed for Yolo object detection system," 2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE). IEEE, 2019: 9-11.

[7] Nguyen D T, Nguyen T N, Kim H, et al, “A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2019, 27(8): 1861-1873.

[8] Chen Y H, Fan C P, Chang R C H, “Prototype of low complexity CNN hardware accelerator with FPGA-based PYNQ platform for dual-mode biometrics recognition," 2020 International SoC Design Conference (ISOCC). IEEE, 2020: 189-190.

[9] Jiang C, Ojika D, Patel B, et al, “Optimized fpga-based deep learning accelerator for sparse cnn using high bandwidth memory," 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2021: 157-164.

[10] Colleman S, Verhelst M, “High-utilization, high-flexibility depth-first CNN coprocessor for image pixel processing on FPGA," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2021, 29(3): 461-471.

[11] Shaydyuk N K, John E B, “FPGA Implementation of MobileNetV2 CNN Model Using Semi-Streaming Architecture for Low Power Inference Applications," 2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). IEEE, 2020: 160-167.

[12] Desavathu P B, Raja A R, Patra S, et al, “Design and Implementation of CNN-FPGA accelerator based on Open Computing Language," 2022 First International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT). IEEE, 2022: 1-4.

[13] Papon C, “SpinalHDL: An alternative hardware description language," FOSDEM. 2017.

[14] Chen T, Moreau T, Jiang Z, et al, “TVM: An automated End-to-End optimizing compiler for deep learning," 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 578-594.

[15] Bachrach J, Vo H, Richards B, et al, “Chisel: constructing hardware in a scala embedded language," DAC Design automation conference 2012. IEEE, 2012: 1212-1221.

[16] Zhang X, Wang J, Zhu C, et al, “DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2018: 1-8.

[17] Yazdanbakhsh A, Brzozowski M, Khaleghi B, et al, “Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks," 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018: 65-72.

[18] Jahanshahi A, Sharifi R, Rezvani M, et al, “Inf4edge: Automatic resource-aware generation of energy-efficient cnn inference accelerator for edge embedded fpgas," 2021 12th International Green and Sustainable Computing Conference (IGSC). IEEE, 2021: 1-8.

[19] Howard A G, Zhu M, Chen B, et al, “Mobilenets: Efficient convolutional neural networks for mobile vision applications," arXiv preprint arXiv:1704.04861, 2017.

[20] Horowitz M, “Computing's energy problem (and what we can do about it)," 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE, 2014: 10-14.

[21] Krishnamoorthi R, “Quantizing deep convolutional networks for efficient inference: A whitepaper," arXiv preprint arXiv:1806.08342, 2018.

[22] Nasser A, Fadel K A, Abbas K O, et al, “An Automated Flow for Configuration and Generation of CNN based AI accelerators for HW Emulation & FPGA Prototyping," 2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS). IEEE, 2021: 1-7.

[23] Fu Y, Wu E, Santhaseelan V, et al, “Embedded vision with int8 optimization on Xilinx devices," WP490 (v1. 0.1), Apr, 2017, 19: 15.

[24] Wu E, Zhang X, Berman D, et al, “A high-throughput reconfigurable processing array for neural networks," 2017 27th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2017: 1-4.

[25] Pestana D, Miranda P R, Lopes J D, et al, “A full featured configurable accelerator for object detection with YOLO," IEEE Access, 2021, 9: 75864-75877.

[26] Song Q, Zhang J, Sun L, et al, “Design and Implementation of Convolutional Neural Networks Accelerator Based on Multidie," IEEE Access, 2022, 10: 91497-91508.