electronics

MDPI

Article

Automatic RTL Generation Tool of FPGAs for DNNs

Seojin Jang [1], Wei Liu [2], Sangun Park [3] and Yongbeom Cho [1,2,*]

[1] Department of Electrical and Electronics Engineering, Konkuk University, Seoul 05029, Korea; seojinygud@konkuk.ac.kr
[2] Deep ET, Seoul 05029, Korea; liuwei1108@deep-et.com
[3] Samsung Electronics, Suwon 16677, Korea; kos8108@naver.com
* Correspondence: ybcho@konkuk.ac.kr

Abstract: With the increasing use of multi-purpose artificial intelligence of things (AIOT) devices, embedded field-programmable gate arrays (FPGA) represent excellent platforms for deep neural network (DNN) acceleration on edge devices. FPGAs possess the advantages of low latency and high energy efficiency, but the scarcity of FPGA development resources challenges the deployment of DNN-based edge devices. Register-transfer level programming, hardware verification, and precise resource allocation are needed to build a high-performance FPGA accelerator for DNNs. These tasks present a challenge and are time consuming for even experienced hardware developers. Therefore, we propose an automated, collaborative design process employing an automatic design space exploration tool; an automatic DNN engine enables the tool to reshape and parse a DNN model from software to hardware. We also introduce a long short-term memory (LSTM)-based model to predict performance and generate a DNN model that suits the developer requirements automatically. We demonstrate our design scheme with three FPGAs: a zcu104, a zcu102, and a Cyclone V SoC (system on chip). The results show that our hardware-based edge accelerator exhibits superior throughput compared with the most advanced edge graphics processing unit.

Keywords: convolutional neural network (CNN); deep learning; hardware/software co-design; FPGA

Citation: Jang, S.; Liu, W.; Park, S.; Cho, Y. Automatic RTL Generation Tool of FPGAs for DNNs. *Electronics* **2022**, *11*, 402. https://doi.org/10.3390/electronics11030402

Academic Editors: Valeri Mladenov, Lucas Lamata, Dah-Jye Lee and Nikolay Hinov

Received: 24 December 2021
Accepted: 26 January 2022
Published: 28 January 2022

1. Introduction

Recent studies have shown that field-programmable gate arrays (FPGA) are promising candidates for deep neural network (DNN) implementation [1–5]. A DNN can be integrated via hardware, rather than via an existing central processing unit (CPU) or graphics processing unit (GPU), thus improving latency and reducing energy consumption. These characteristics exemplify FPGAs for DNN-based applications in cloud and edge computing; as a result, FPGAs have been rapidly adopted for DNN acceleration. Internet of things (IOT) applications have stringent requirements in the fields of automatic driving, safety, and monitoring; complex DNN models must produce quality results with minimal delay and power consumption, while not exceeding resource constraints [6].

Embedded FPGAs are the most attractive candidate to integrate machine learning with IOT [7] because they are energy efficient and affordable. However, development resource scarcity makes the design and deployment of FPGA DNN accelerators challenging.

Our study alleviates several challenges in the field with emphasis on the following main ideas:

(1) This study proposes DNN implementation via an auto-generated automation tool, which maps the DNN design process from a deep learning framework to FPGA. As the structure of DNNs changes rapidly, it is difficult for their hardware design to keep up with the software design. Therefore, our proposition allows users to perform resource allocation and optimization during register-transfer level (RTL) design when deploying a DNN on FPGAs.

(2) A DNN model was generated according to developer performance guidelines during the design process. An automatic DNN model optimization engine is proposed based on this process, by implementing a long short-term memory (LSTM) that can effectively generate DNN models that meet the performance requirements of FPGA design.
(3) Highly optimized RTL network components can be automatically generated for building DNN layers. Since different FPGA manufacturers use different intellectual property (IP) cores for multiplication in digital signal processing (DSP), we propose a multiplier optimization for DNN processing elements (PEs) with improved energy consumption. The engine can be configured to provide the best performance within the constraints of the FPGA.

2. Related Work

Designing a DNN model and FPGA accelerator requires meticulous research, which is often carried out independently. Machine learning experts either manually design DNNs or employ automatic design via a recurrent neural network (RNN) and reinforcement learning. In FPGA acceleration, traditional and Winograd convolutions are combined to run DNNs on embedded FPGAs [8]. Aydonat et al. showed that the Winograd-based solution could be implemented on FPGAs, reducing the need for multiplication in a DNN [9].

Studies involving platform-based DNN search methods [10,11] improve upon previously published results. However, the authors of these studies only considered DNN inference delay on the CPU and GPU, neglecting DNN inference delay on FPGAs.

New techniques such as quantization [8,12] and model compression [13,14] are also used in the FPGA DNN accelerator to reduce the size of the DNN model and diminish latency in the DNN inference process. However, these methods may be limited by the DNN model itself, as not all DNN models can be compressed. In addition, the compressed DNN model does not necessarily meet the real-time performance constraints of target artificial IOT (AIOT) platforms.

Automation tools to quickly deploy DNNs into FPGAs have been investigated by other researchers [15]. They adopted a unified RTL design for the convolution (conv) layer to permit different network configurations, but the fully connected (FC) layer was not implemented on the FPGA. The proposed RTL template mapped DNN onto FPGAs automatically. However, the DLL used a unified computing unit (for example, a fixed-size computing engine), which yields reduced computing performance when compared with a dedicated design for different networks [16,17]. In these studies, the high-level synthesis (HLS) template was used to map DNNs to FPGAs automatically, significantly improving the RTL size [18].

3. Materials and Methods

3.1. Proposed Automation Flow

We propose a space exploration model to find the optimal accelerator design that balances the DNN network's constraints and the FPGA's performance. As shown in Figure 1, we generated the DNN accelerator via a three-step procedure that included a network analyzer, required-performance-based PE optimization, and hardware description language (HDL) auto-generation. The steps are described as follows:

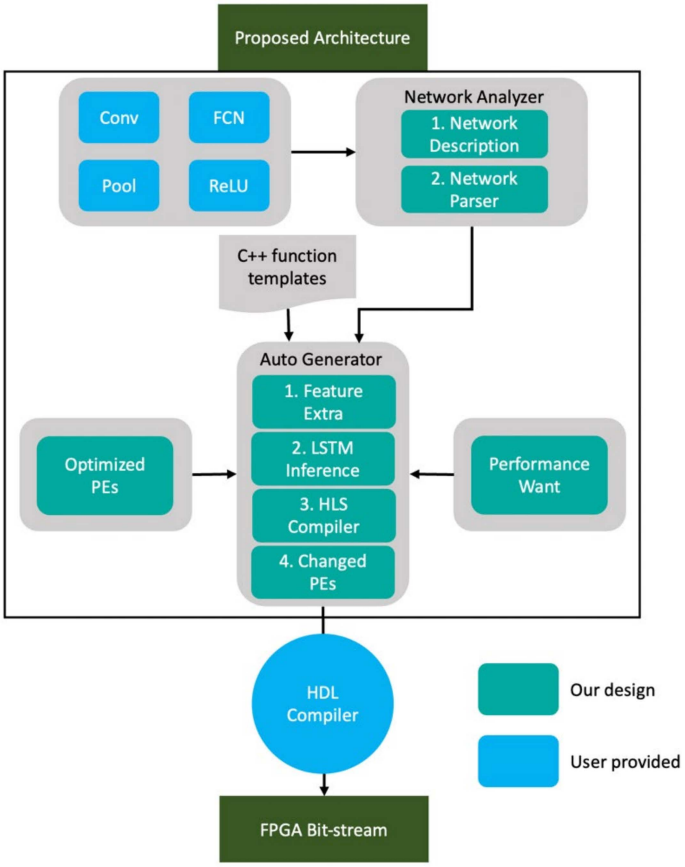

Figure 1. Proposed system.

(1) Developers first use a deep learning open neural network exchange (ONNX) framework to design and train the DNN target network during the design process. After training, the DNN network definition file is passed to the DNN network analyzer [19].
(2) A network analysis decomposes network layers from the network model, such as the conv, pooling, and FC layers. Then, it maps them to the HLS template. The network analyzer retains the operation logic and bit size in the conv, pooling, and FC layers to ensure that logic remains as designed.
(3) In the third step, the LSTM-based generator automatically designs the optimized DNN HLS.

We optimized the DNN design in two ways. First, we redesigned the PE so that energy consumption is optimized in the automatically generated neural network. The proposed PE also supports N-bit calculations, permitting the automatic quantization of the network into N-bits. Second, we implemented feature value extraction in the HLS C/C++ inference code and used this feature value to train the LSTM. This automatically generates the DNN-RTL design with the highest degree of parallelism.

3.2. N-Bit Vedic Multiplier for DNN PEs

PEs are an essential tool in DNN operations. However, there are two limitations when using an embedded multiplier or embedded DSP block to construct a PE: PE size is fixed, and PE availability is limited. In addition, they have a fixed position on the FPGA itself and often have bit size asymmetry (the 18 × 25 multiplier of Xilinx FPGAs, for example) [20]. Therefore, we designed a Vedic multiplier-based N-bit processing element to overcome the limitations above. Vedic multipliers have a higher speed, smaller area, and lower power consumption than FPGA-provided multipliers employing convolution operations. Our PEs have the added advantage of reduced memory usage.

Vedic Mathematics demonstrates low latency for multiplication and employs a unique technique based on 16 Sutras [21]. The specific method is found in Urdhva Tiryakbhyam, one of the 16 Sutras, and performs multiplication in a crosswise and vertical manner. Thus, the method allows numerical computations at a high pace by generating partial products and sums in a single iteration [22].

The Vedic multiplier architecture was used in this study, employing smaller modules to permit the multiplication of large numbers (N × N bits). We used a 2 × 2 Vedic multiplier as a fundamental building module from which higher multipliers are assembled, as shown in Figure 2. Adders were used to perform intermediate computations, and consequently optimized adder arrangements can improve multiplier efficiency [23]. Improving intermediate adder efficiency will enhance multiplier efficiency while reducing latency and size [24]. The ripple carry adder (RSA) design is a standard adder design that consists of a series of full adders. All bits must be queued, as the carry out bit of each full adder is used for the next bit operation. While this design is easy to implement, it is inefficient and slow. A carry look-ahead adder (CLAA) can compensate for the limitations of RSA by generating and propagating carries. The speed of a CLAA is balanced by the increased complexity; the number of gates and fan-in increases as the most significant digit increases. A prefix adder performs parallel operations over several stages; the Kogge–Stone adder (KSA) is an example of this type of adder. The KSA is a parallel prefix form of the CLAA and is widely considered to be the fastest adder in the industry [23]. Thus, we designed the Vedic multiplier with the KSA as shown in Figure 3.

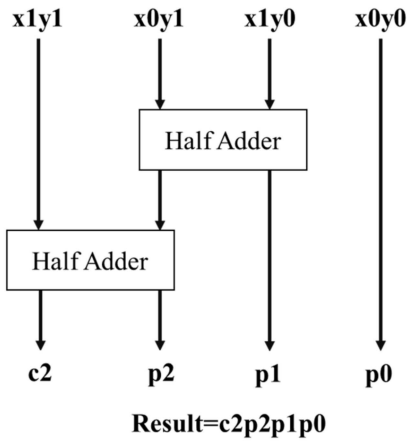

Figure 2. Block Diagram of 2 × 2 Vedic Multiplier.

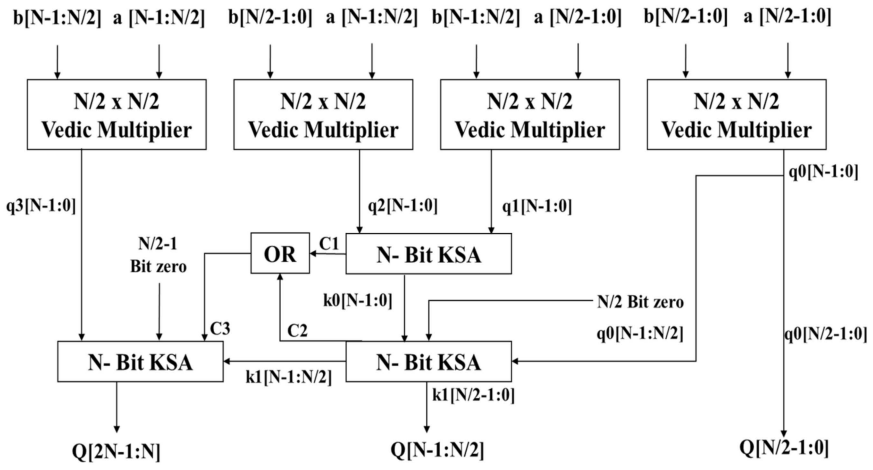

Figure 3. Block Diagram of Vedic multiplier using KSA.

A simulation was performed using Synopsys' Verdi simulator. The RTL simulation results for a 32 bits × 32 bits Vedic multiplier are shown in Figure 4, in which Input 1 = 36, Input 2 = 48, and Output = 1728. The synthesis was performed using a Synopsys Design Compiler. The above design was implemented in Verilog code using VCS. The RTL code was synthesized using Design Compiler in 65 nm technology. Area, speed, and power reports were compared with other methods, as listed in Table 1. Additionally, Table 2 lists the FPGA resources consumed in the implementation of the 32 × 32 multiplier. We used Vivado 2019.4 as the compiler and performed a demonstration on a ZCU102 board. Compared with the multiplier resources provided by Xilinx, there was a 31% reduction in Slice, a 16% reduction in input output block (IOB), and a 13% increase in lookup table (LUT).

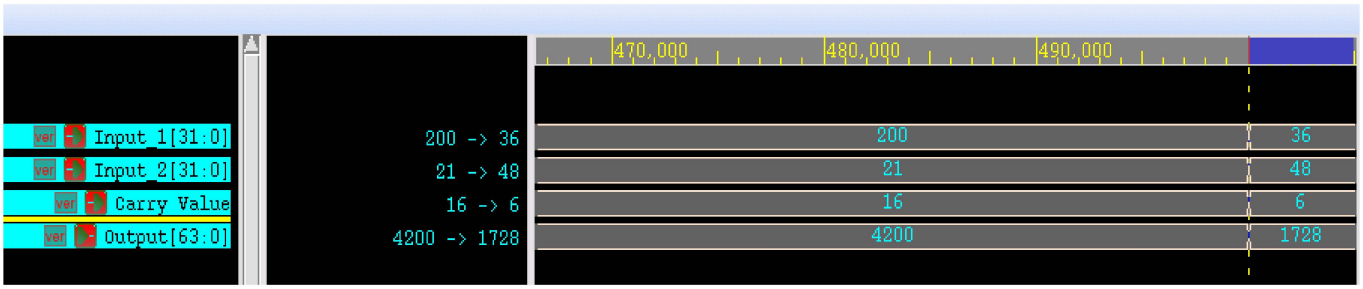

Figure 4. Simulation results of 32 × 32 Vedic multiplier.

Table 1. Analysis report of conventional multiplier and proposed Vedic multiplier.

Parameter	Conventional Multiplier	Conventional Vedic Multiplier	Proposed Vedic Multiplier
Power (μW)	26.351	27.235	23.33
Speed (ns)	6.213	5.327	4.1
Area (μm^2)	3567	4283	3958

Table 2. Resource utilization of 32 × 32 multiplier.

Parameter	Slice	LUT	DSP	IOB
Xilinx IP	415	93	2	102
Ours	287	107	0	86

3.3. Performance-Based Auto-Generator

Computation and communication are the two primary constraints used to improve the throughput of an accelerator. The roofline performance model quantitatively analyzes the computed throughput and required memory bandwidth for any potential solution to a convolutional neural network (CNN) design on an FPGA platform. The roofline calculation density refers to the amount of calculation required by a program per unit of memory access, with units of *FLOPs*/*byte*. The calculation formula is as follows:

$$I = \left(\frac{FLOPs}{byte} \right) \tag{1}$$

The roofline model is used to evaluate the upper bound of a program's performance on hardware, and is displayed using the following Figure 5:

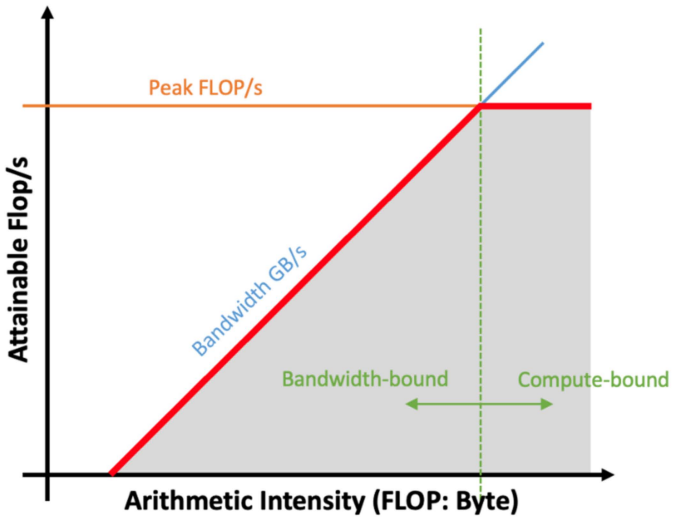

Figure 5. Roofline model.

When the calculation density I of the inference program is small, the program fetches more memory per calculation. In other words, the memory bandwidth limits the hardware design. The upper bound of program performance is expressed as a diagonal line in Figure 5, where the slope represents memory bandwidth. The greater the calculation density, the higher the upper bound of the program's achievable speed, while always using the maximum memory bandwidth. On the contrary, if the calculation density I is large, the program performance is limited by the maximum calculation peak of the hardware (called a calculation-intensive program). Beyond the maximum calculation peak, performance is limited by the hardware computing power, which is expressed as a horizontal line in Figure 5. The calculation speed is no longer affected by the calculation density in this regime. The calculation speed and the memory bandwidth are maximized at the intersection of the two lines; this intersection point denotes the full use of hardware resources.

Not all operators have the same computational properties in a deep learning network. That is, they may use more or less memory while performing similar calculations and are not in the same position on the roofline. Operators in deep learning networks can be classified according to computational density; conv, FC, and deconv operators are calculation-intensive operators, while ReLU, EltWise Add, Concat, etc., are memory-access-intensive operators. The same operator will also change the calculation density or change its properties owing to different parameters. For example, increasing the group or reducing the input channels of Conv will reduce the calculation density under the premise that other parameters remain unchanged.

For memory-intensive operators, the inference time has a linear relationship with memory access. By contrast, inference time has a linear relationship with calculation for computationally intensive operators. A deep learning network is often composed of a mixture of memory-intensive operators and computationally intensive operators, in which the operator attributes change. We cannot design the hardware statistically to completely leverage hardware performance.

Therefore, we propose an LSTM-based method to maximize performance-based automatic DNN structure generation. First, we implemented a method for extracting HLS features in C/C++. Then, an LSTM was designed, and the extracted features were used to predict new HLS parallel methods. We used an abstract syntax tree (AST) model to extract structure, loop nesting, variables, operators, and other specific values from HLS code. Figure 6 shows three components (parallel structure, memory access, and operands) of the vector.

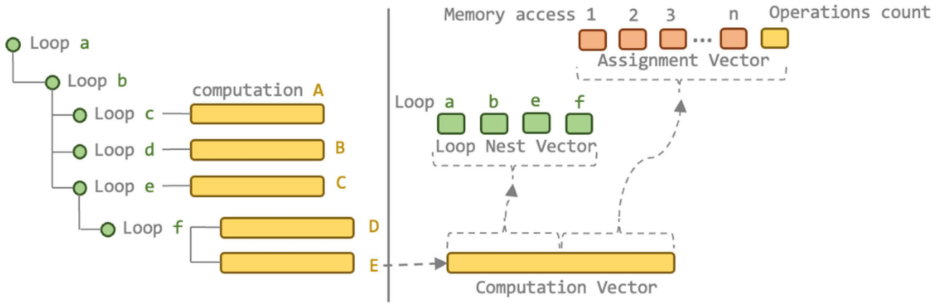

Figure 6. Parallel structure, memory access, and operand expression of the DNN structure.

Loop nesting codes are represented by arranging loops from the outermost to the innermost. Thus, features extracted from the loop nest can be stored as the start and end number, as shown in Figure 7.

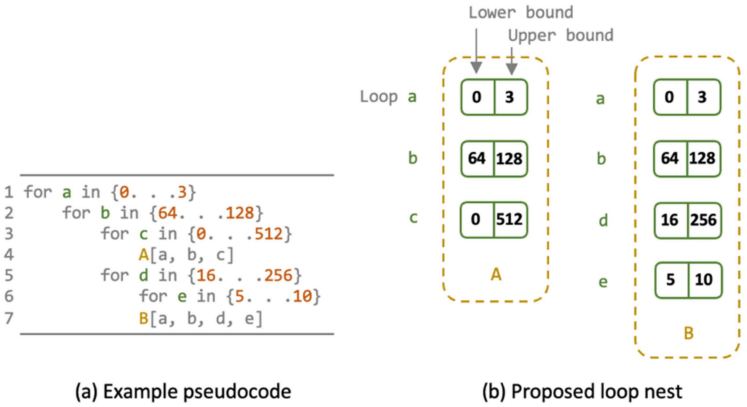

Figure 7. Loop stored method.

We added tags to the loop to distinguish each loop nest level, as shown in Figure 8. Adding tags and merging cycles can avoid repeated use of loops.

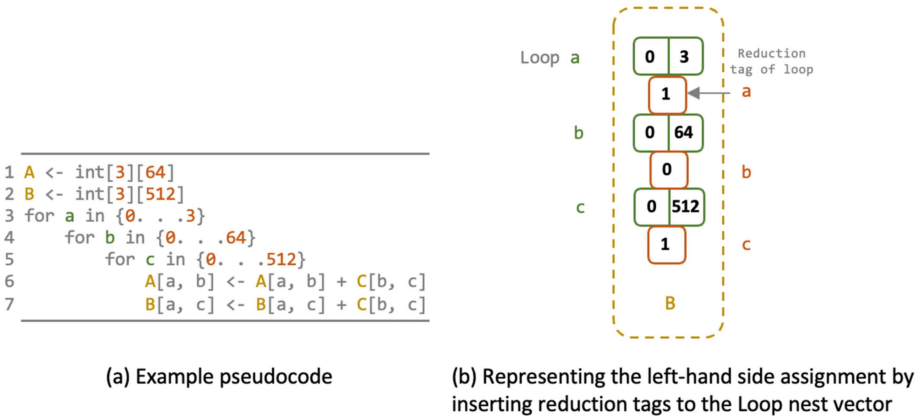

Figure 8. Add tag and merge loop nest.

Figures 6 and 7 show the memory access, operand expression, and loop storage of the parallel structure as parameters. In Figure 8, we characterize loop optimization by type, parameter, and point before hardware synthesis. The loop represented in the figure uses the previous loop size as the marking method because the loop level and operation affect the calculation results of the algorithm, as shown in Figure 9.

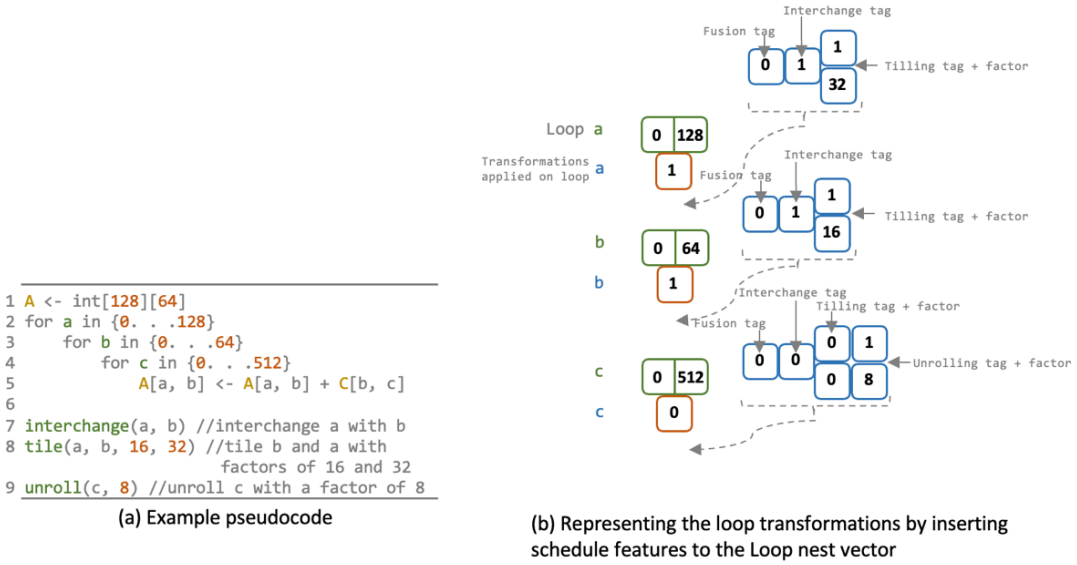

Figure 9. Loop nest optimization pseudocode and representation.

To design an automatic scheduler, we extracted specific values (Figure 9) that determine which loop to use for Tile and Unrolling HLS pragma. As shown in Figure 10, different parallel optimizations are performed to obtain different scheduling methods in the generated HLS code. Then, the scheduler is transformed into a re-expression of the above statement through the translation method. In this process, different schedulers will synthesize various hardware designs (hardware size and latency vary between synthesized designs). These data are then used for further training via the LSTM, as shown in Figure 11.

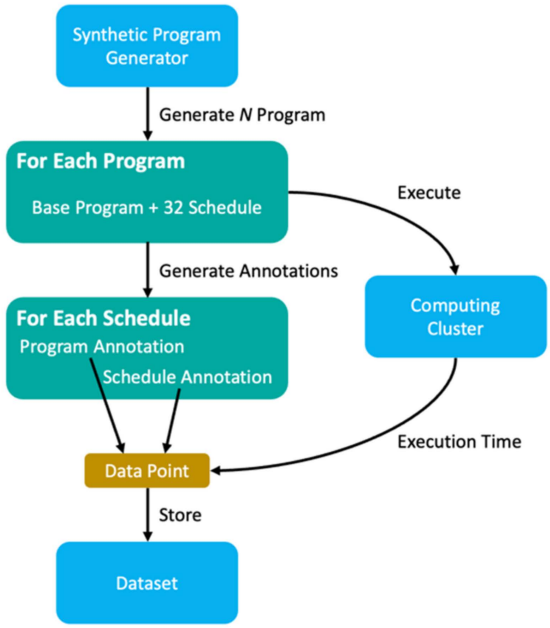

Figure 10. Workflow of proposed method.

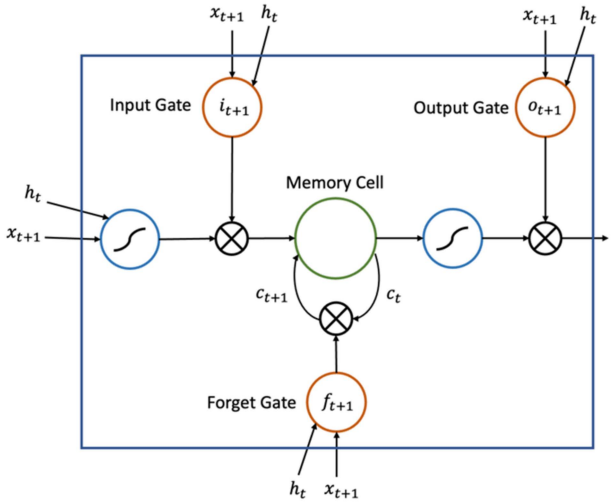

Figure 11. Long short-term memory training unit.

Design space exploration identifies the most optimal design with the highest throughput under the performance and resource modeling from LSTM predictions. However, the designed operating frequency is difficult to predict. Therefore, we developed a two-stage process as described in Figure 12, where we first used the proposed analysis model to filter the design space into a small set of candidates with a similar and predetermined clock frequency. Then, we generated hardware to obtain a design with the best onboard performance. The design flow framework is button-based, providing an intuitive DNN program that users write to generate an executable system on FPGA automatically. The user need not specify the nested loops of the DNN layer.

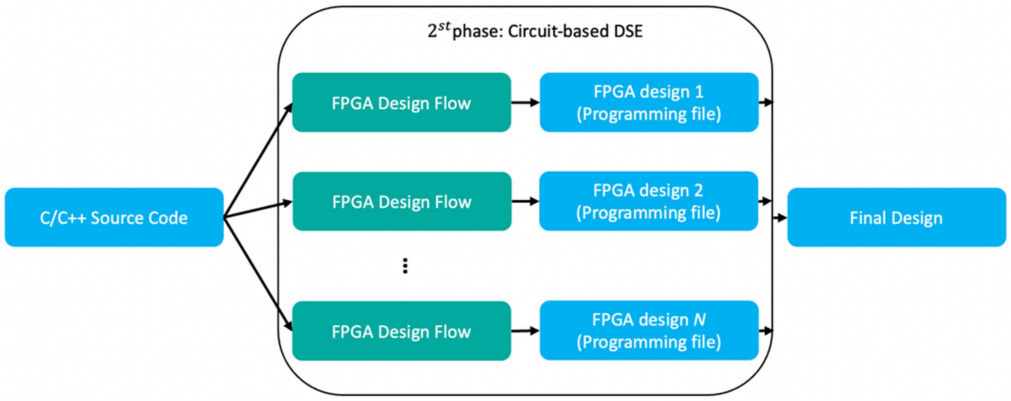

Figure 12. Two-stage design space exploration.

4. Results

We used the proposed scheme to build four DNNs: ResNet, MobileNet, DensNet, and VGG with an input size of 224 × 224. We compared it with the latest embedded GPUs, TX2 and TX1. We tried to use a minimum batch size owing to the real-time requirements of edge applications. We sought an example FPGA board with few gates and chose the Zynq family that was used by Oiu et al. [12] and Suda et al. [25] for comparison. Based on the experimental results summarized in Table 3, we observe that our design in Xilinx zcu104, zcu102, and Intel Cyclone V SOC (System on chip) provided higher efficiency than the TensorRT inference solution based on Nvidia Jetson TX2.

Table 3. Evaluation on FPGA (batch size = 1 for Fix16).

	ZCU 104 (ms)	ZCU 102 (ms)	Cyclone V SoC (ms)	Jetson Tx1 (ms)	Jetson Tx2 (ms)
ResNet18	1.78	2.16	4.1	21	14.7
VGG-16	12.9	15.52	29.1	151	105.7
MobileNetV2	1.77	2.15	3.95	20.5	14.3
DenseNet-121	5.67	6.79	12.8	66.4	46.4

We compared the accelerator design with those from two related studies in Table 4 [12,25]. We examined two aspects of performance: (1) peak CONV layer performance and (2) overall performance of all CONV layers. Our design performed significantly better than those published in previous studies.

Table 4. Comparison with other FPGA work.

	Oiu [12]	Suda [25]	Ours		
CNN Models			**VGG**		
Device	Zynq 7Z45	Stratix V GSD8	ZCU 104	ZCU 102	Cyclone V SoC
Precision	Fixed 16 bit	Fixed 16 bit	Fixed 16 bit	Fixed 16 bit	Fixed 16 bit
Overall CONV GOPs	187.8	136.5	304.2	253.3	144.7
Peak CONV	254.8	-	368.5	304.7	260.6

We used VGG16, which has 13 convolutional and three fully connected layers. Table 5 compares our proposed method with optimized CPU single instruction multiple data (SIMD) and GPU solutions. We used actual achieved performance (GOPS) as the standard measurement for fair comparison. We also used a zcu02, a zcu04, and a Kirin 970 CPU (with Arm computing library optimized) to achieve approximately 46 times the acceleration and 55 times the performance improvement.

Table 5. Comparison with CPU/GPU edge computing.

Platform	**CPU (SIMD)**	**GPU**		**FPGA**		
Device	Kirin 970	Jetson Tx1	Jetson Tx2	ZCU 104	ZCU 102	Cyclone V SoC
Precision	Float32	Float32	Float32	Fixed 16	Fixed 16	Fixed 16
Batch size	1	1	1	1	1	1
Latency	720.9	151	105.7	12.9	15.52	29.1
Speedup	1×	4.77×	6.82×	55.88×	46.44×	24.77×
Power (Watt)	8	10	7.5	6.3	7.6	8.2

5. Conclusions

To meet the requirements of real-time IoT and AIoT applications, performance must be improved. Currently, cloud servers take care of the high computation requirements of complex AI applications. However, real-time computations depend on embedded equipment to avoid network delay. IoT devices are not known for powerful computations or real-time calculations. With the rising implementation of AI networks for tasks that require high precision or many variables, device localization presents a challenge. FPGA designs are the answer to these problems, as they are highly energy efficient and low cost. However, developing FPGAs is time consuming and labor intensive. In addition, they may only work within a single AI network, with further customization requiring additional time. Developers are searching for a tool that can "automatically generate" AI networks on FPGAs, thereby reducing time consumption and generating the AI networks at a hardware level. Therefore, we propose a high-performance deep learning inference optimizer with low-latency, high-throughput deployment inference for AI applications. Experimental results showed that the performance of the proposed method increased by

approximately 50 and 5 times compared with CPU SIMD (single instruction, multiple data) and GPU-based edge approaches, respectively.

Author Contributions: Date curation, S.J. and S.P.; Formal analysis, W.L.; Founding acquisition, Y.C.; Investigation, Y.C.; Resources, W.L.; Software, S.J. and W.L.; Writing—original draft, S.J.; Supervision, Y.C.; Validation, Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Korea Evaluation Institute of Industrial Technology (KEIT) under the Industrial Embedded System Technology Development (R&D) Program 20016341. The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015.
2. Kim, T.; Park, S.; Cho, Y. Study on the Implementation of a Simple and Effective Memory System for an AI Chip. *Electronics* **2021**, *10*, 1399. [CrossRef]
3. Zhang, X.; Wang, J.; Zhu, C.; Lin, Y.; Xiong, J.; Hwu, W.M.; Chen, D. Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas. In Proceedings of the 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Diego, CA, USA, 5–8 November 2018.
4. Li, Q.; Zhang, X.; Xiong, J.; Hwu, W.M.; Chen, D. Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS. In Proceedings of the 24th Asia and South Pacific Design Automation Conference, Tokyo, Japan, 21–24 January 2019.
5. He, D.; He, J.; Liu, J.; Yang, J.; Yan, Q.; Yang, Y. An FPGA-Based LSTM Acceleration Engine for Deep Learning Frameworks. *Electronics* **2021**, *10*, 681. [CrossRef]
6. Qi, X.; Liu, C. Enabling Deep Learning on IoT Edge: Approaches and Evaluation. In Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 25–27 October 2018. [CrossRef]
7. Zhang, X.; Ramachandran, A.; Zhuge, C.; He, D.; Zuo, W.; Cheng, Z.; Rupnow, K.; Chen, D. Machine learning on FPGAs to face the IoT revolution. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017.
8. Wang, J.; Lou, Q.; Zhang, X.; Zhu, C.; Lin, Y.; Chen, D. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL), Dublin, Ireland, 27–31 August 2018.
9. Aydonat, U.; O'Connell, S.; Capalija, D.; Ling, A.C.; Chiu, G.R. An opencl™ deep learning accelerator on arria 10. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017.
10. Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; Sandler, M.; Howard, A.; Le, Q.V. Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019.
11. Cai, H.; Zhu, L.; Han, S. Proxylessnas: Direct neural architecture search on target task and hardware. In Proceedings of the 2019 7th International Conference on Learning Representation (ICLR), New Orleans, LA, USA, 6–9 May 2019.
12. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016.
13. Han, S.; Kang, J.; Mao, H.; Hu, Y.; Li, X.; Li, Y.; Xie, D.; Luo, H.; Yao, S.; Wang, Y.; et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017.
14. Zhang, M.; Li, L.; Wang, H.; Liu, Y.; Qin, H.; Zhao, W. Optimized Compression for Implementing Convolutional Neural Networks on FPGA. *Electronics* **2019**, *8*, 295. [CrossRef]
15. Zeng, H.; Chen, R.; Zhang, C.; Prasanna, V. A framework for generating high throughput CNN implementations on FPGAs. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018.
16. Sharma, H.; Park, J.; Mahajan, D.; Amaro, E.; Kim, J.K.; Shao, C.; Mishra, A.; Esmaeilzadeh, H. From high-level deep neural models to FPGAs. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016.
17. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017.

18. Guan, Y.; Liang, H.; Xu, N.; Wang, W.; Shi, S.; Chen, X.; Sun, G.; Zhang, W.; Cong, J. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017.
19. Lin, W.F.; Tsai, D.Y.; Tang, L.; Hsieh, C.T.; Chou, C.Y.; Chang, P.H.; Hsu, L. ONNC: A compilation framework connecting ONNX to proprietary deep learning accelerators. In Proceedings of the 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), Hsinchu, Taiwan, 18–20 March 2019.
20. Kumm, M.; Abbas, S.; Zipf, P. An efficient softcore multiplier architecture for Xilinx FPGAs. In Proceedings of the 2015 IEEE 22nd Symposium on Computer Arithmetic, Lyon, France, 22–24 June 2015.
21. Tiwari, H.D.; Gankhuyag, G.; Kim, C.M.; Cho, Y.B. Multiplier design based on ancient Indian Vedic Mathematics. In Proceedings of the 2008 International SoC Design Conference, Busan, Korea, 24–25 November 2008.
22. Karthik, V.K.; Govardhan, Y.; Reddy, V.K.; Praveena, K. Design of Multiply and Accumulate Unit using Vedic Multiplication Techniques. *Int. J. Sci. Res.* **2013**, *4*, 756.
23. Jayanthi, A.N.; Ravichandran, C.S. Comparison of performance of high speed VLSI adders. In Proceedings of the 2013 International Conference on Current Trends in Engineering and Technology (ICCTET), Coimbatore, India, 3 July 2013.
24. Akhter, S.; Saini, V.; Saini, J. Analysis of Vedic multiplier using various adder topologies. In Proceedings of the 2017 4th International Conference on Signal Processing and Integrated Networks (SPIN), Noida, India, 2–3 February 2017.
25. Suda, N.; Chandra, V.; Dasika, G.; Mohanty, A.; Ma, Y.; Vrudhula, S.; Seo, J.S.; Cao, Y. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016.