electronics

MDPI

Article

CNN2Gate: An Implementation of Convolutional Neural Networks Inference on FPGAs with Automated Design Space Exploration

Alireza Ghaffari * and Yvon Savaria

Electrical Engineering Department, Polytechnique Montreal, 2500 Edouard Montpetit Blvd, Montreal, QC H3T 1J4, Canada; yvon.savaria@polymtl.ca

* Correspondence: seyed-alireza.ghaffari@polymtl.ca

Received: 23 November 2020; Accepted: 16 December 2020; Published: 21 December 2020

Abstract: Convolutional Neural Networks (CNNs) have a major impact on our society, because of the numerous services they provide. These services include, but are not limited to image classification, video analysis, and speech recognition. Recently, the number of researches that utilize FPGAs to implement CNNs are increasing rapidly. This is due to the lower power consumption and easy reconfigurability that are offered by these platforms. Because of the research efforts put into topics, such as architecture, synthesis, and optimization, some new challenges are arising for integrating suitable hardware solutions to high-level machine learning software libraries. This paper introduces an integrated framework (CNN2Gate), which supports compilation of a CNN model for an FPGA target. CNN2Gate is capable of parsing CNN models from several popular high-level machine learning libraries, such as Keras, Pytorch, Caffe2, etc. CNN2Gate extracts computation flow of layers, in addition to weights and biases, and applies a "given" fixed-point quantization. Furthermore, it writes this information in the proper format for the FPGA vendor's OpenCL synthesis tools that are then used to build and run the project on FPGA. CNN2Gate performs design-space exploration and fits the design on different FPGAs with limited logic resources automatically. This paper reports results of automatic synthesis and design-space exploration of AlexNet and VGG-16 on various Intel FPGA platforms.

Keywords: automated high-level synthesis; Convolutional Neural Network (CNN); design-space exploration; FPGA; hardware optimization; hardware-aware FPGA fitter; Open Neural Network Exchange Format (ONNX); reinforcement learning; Register Transfer Level (RTL)

1. Introduction

The impact of machine learning and deep learning is rapidly growing in our society, due to their diverse technological advantages. Convolutional neural networks (CNNs) are among the most notable architectures that provide a very powerful tool for many applications, such as video and image analysis, speech recognition, and recommender systems [1]. On the other hand, CNNs require considerable computing power. In order to better satisfy some given requirements, it is possible to use high-performance processors, like graphic processing units (GPUs) [2]. However, GPUs have some shortcomings that limit their usability and suitability in day-to-day mission-critical and real-time scenarios. The first downside of using GPUs is their high power consumption. This makes GPUs hard to use in robotics, drones, self-driving cars, and Internet of Things (IoTs), while these fields can highly benefit from deep learning algorithms. The second downside is the lack of external Inputs/Outputs (I/Os). GPUs are typically accessible through some PCI-express bus on their host

computer. The absence of direct I/Os makes it hard to use them in mission-critical scenarios that need prompt control actions.

In particular, the floating-point number representation of CPUs and GPUs is used for most deep learning algorithms. However, it is possible to benefit from custom fixed-point numbers (quantized numbers) in order to reduce the power consumption, circuit footprint, and increase the number of compute engines [3]. It was proven that many convolutional neural networks can work with eight-bit quantized numbers or less [4]. Hence, GPUs waste significant computing power and electrical power consumption when performing inference in most deep learning algorithms. A better balance can be restored by designing application specific computing units while using quantized arithmetic. Quantized numbers provide other benefits, such as memory access performance improvements, as they allocate less space in the external or internal memory of compute devices. Memory access performance and memory footprint are particularly important for hardware designers in many low-power devices. Most of the portable hardware devices, such as cell-phones, drones, and internet of things (IoT) sensors, require memory optimization due to of the limited resources available on these devices.

Field Programmable Gate Arrays (FPGA) can be used in these scenarios in order to tackle the problems that are caused by limitations of GPUs without compromising the accuracy of the algorithm. Using quantized deep learning algorithms can solve the power consumption and memory efficiency issues on FPGAs as the size of the implemented circuits shrinks with quantization. In [5], the authors reported that the theoretical peak performance of six-bit integer matrix multiplication (GEMM) in the Titan X GPU is almost 180 GOP/s/Watt, while it can be as high as 380 GOP/s/Watt for Stratix 10, and 200 GOP/s/Watt for Arria 10 FPGAs. This means that high end FPGAs are able to provide comparable or even better performance per Watt than the modern GPUs. FPGAs are scalable and configurable. Thus, a deep convolutional neural network can be configured and scaled to be used in a much smaller FPGA in order to reduce the power consumption for mobile devices. The present paper investigates various design-space exploration methods that can fit a desired CNN to a selected FPGA with limited resources. Having massive connectivity through I/Os is natural in FPGAs. Furthermore, FPGAs can be flexibly optimized and tailored to perform various applications in industrial and real-time mobile workloads [6]. FPGA vendors, such as Intel, offer a range of commercialized System-on-Chip (SoC) devices that integrate both processor and FPGA fabric into a single chip (see, for example, [7]). These SoCs are widely used on mobile devices and they make the use of application-specific processors obsolete in many cases .

Several research contributions address ghd architecture, synthesis, and optimization of deep learning algorithms on FPGAs [8–12]. Nonetheless, little work was done regarding integrating the results into a single tool. An in depth literature review in Section 2 exposes that tools, such as [13,14], provide neither integrated design space exploration nor generic CNN model analyzer. In [11,12,15], the authors specifically discuss the hardware design and not the automation of the synthesis. In [9], fitting the design automatically in different FPGA targets is not discussed. This presents a major challenge, as there are many degrees of freedom in designing a CNN. There is no fixed architecture for CNN, as a designer can choose as many convolution, pooling, and fully connected layers as needed to reach the desired accuracy. In addition, there is a need for design-space exploration methods that optimize resource utilization on FPGA.

A number of development environments exist for the description and architectural synthesis of digital systems [16,17]. However, they are designed for general purpose applications. This means that there is a great opportunity to make a library that uses these hardware design environments to implement deep learning algorithms. Alternately, there are also many other development frameworks/libraries for deep learning in Python language. Some notable libraries for deep learning are Pytorch, Keras, Tensorflow, and Caffe. Another challenge would be designing a synthesis tool for FPGA that can accept models that are produced by all of the aforementioned Python libraries and many more available for machine learning developers.

The present paper proposes methods for tackling research challenges that are related to the integration of high-level synthesis tools for convolutional neural networks on FPGAs. The contributions of the present paper are:

1. Generalized model analysis

   Most of the previous implementations of the CNNs on FPGA fall short in supporting as many machine learning libraries as possible. CNN2Gate benefits from a generalized model transfer format, called ONNX (Open Neural Network eXchange format) [18]. ONNX helps hardware synthesis tools to be decoupled from the framework in which a specific CNN was designed. Using ONNX in CNN2Gate provides us the ability to focus on hardware synthesis without being concerned by details of specific machine learning tools. CNN2Gate integrates an ONNX parser that extracts the computation data-flow, as well as weights and biases from the ONNX representation of a CNN model. It then writes these data in a format that is more usable with hardware synthesis workflows.

2. Automated high-level synthesis tool

   Several hardware implementation aspects of machine learning algorithms are not part of the skill set of many machine learning engineers and computer scientists. High-level synthesis is one of the solutions that aims at improving hardware design productivity. CNN2Gate offers a high-level synthesis workflow that can be used to implement CNN models on FPGA. CNN2Gate is a Python library that parses, gets the design synthesized using vendor's OpenCL tools, and runs a CNN model automatically. Its embedded procedures aim to achieve the best throughput performance. Furthermore, CNN2Gate eliminates the need for FPGA experts to manually implement the CNN model targeting FPGA hardware during the early stages of the design. Having the form of a Python library, CNN2Gate can be directly exploited by machine learning developers in order to implement a CNN model on FPGAs. CNN2Gate is built around an available open-source tool, notably PipeCNN [8], which exploits the capabilities of existing design tools to use OpenCL kernels from which high-level synthesis can be performed.

3. Hardware-aware design-space exploration

   An important aspect of design-space exploration is to try choosing design parameters that achieve some desired performance prior to the generation of physical design. CNN2Gate provides a design-space exploration tool that is able to adjust the level of parallelism of the algorithm to fit the design on various FPGA platforms. The design-space exploration methods that are proposed here use estimations of hardware resource requirements (e.g., DSPs, lookup tables, registers, and on-chip memory) in order to fit the design. These estimates are obtained by invoking the first stage of the synthesis tool that was provided by the FPGA vendor and receives back the estimated hardware resource utilization. In the next step, the tool tunes the design parameters according to the resource utilization feedback and iterates again to obtain the new hardware resource utilization. We have implemented three algorithms to undertake design-space exploration. The first algorithm is based on brute-force in order to check all possible parameter values. The second algorithm is based on a reinforcement learning (RL) agent that explores the design space with a set of defined policies and actions. Finally, the third algorithm uses the hill-climbing method in order to tackle the problem of design-space exploration in large convex design-spaces. The advantages and disadvantages of these three exploration algorithms will be discussed in the corresponding sections.

Note that CNN2Gate emphasizes software/hardware co-design methods. The goal of this paper is not to compare the performance of FPGAs and GPUs, but it explores the possibility of designing end-to-end generic frameworks that can leverage high-level model descriptions in order to implement FPGA-based CNN accelerators without human intervention. As shown in Figure 1, CNN2Gate is a Python library that can be used in order to perform inference of CNNs on FPGAs that is capable of:

- parsing CNN models;
- extracting the computation flow of each layer;
- extracting weights and biases of each kernel;
- applying post-training quantization values to the extracted weights and biases;
- writing this information in the proper format for hardware synthesis tools;
- performing design space exploration for various FPGA devices using a hardware-aware smart algorithm; and,
- synthesizing and running the project on FPGA using the FPGA vendor's synthesis tool.

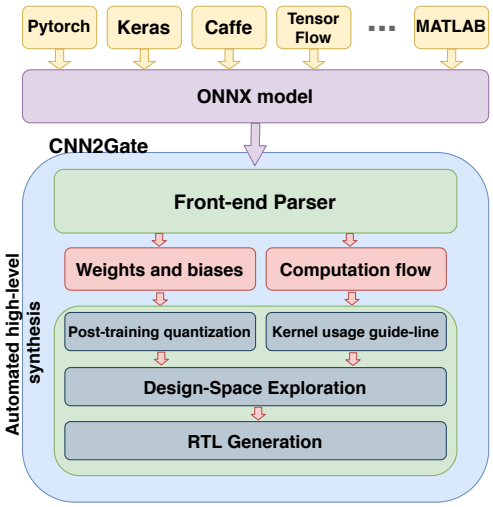

Figure 1. The CNN2Gate overall architecture comprising a front-end parser (Open Neural Network eXchange format (ONNX) parser), a design-space exploration module, and leverages automated high-level synthesis.

The rest of the paper is organized, as follows. Section 2 reviews the related works. Section 3 reviews the most relevant background knowledge on convolutional neural networks. Section 4 elaborates on how CNN2Gate extracts the computation flow, configures the kernels, and executes design space exploration. Subsequently, Section 5 reports some results and compares them to other existing implementations.

2. Related Works

A great deal of research was conducted on implementing deep neural networks on FPGAs. Among those researches, hls4ml [14], fpgaConvNet [9], and Caffeine [15] are the most similar to the present paper. hls4ml is a companion compiler package for machine learning inference on FPGA. It translates open-source machine learning models into high-level synthesizable (HLS) descriptions. hls4ml was specifically developed for an application in particle physics, with the purpose of reducing the development time on FPGA. However, as stated in the status page of the project [19], the package only supports Keras and Tensorflow for CNNs and the support for Pytorch is in development. In addition, to the best of our knowledge, hls4ml does not offer design-space exploration. FpgaConvNet is also an end-to-end framework for the optimized mapping of CNNs on FPGAs. FpgaConvNet uses a symmetric multi-objective algorithm in order to optimize the generated design for either throughput, latency, or multi-objective criteria (e.g., throughput and latency). The front-end parser of fpgaConvNet can analyze models that are expressed in the Torch and Caffe machine-learning libraries. Caffeine is also a software-hardware co-design library that directly synthesizes Caffe models comprising convolutional layers and fully connected layers for FPGAs. The main differences between CNN2Gate and other cited works are in three key features. First, as explained in Section 4.1, CNN2Gate leverages a model transfer layer (ONNX), which automatically brings support for most machine-learning Python libraries without bounding the user to a specific machine-learning library. Second, CNN2Gate is based

on OpenCL, unlike hls4ml and fpgasConvNet, which are based on C++. Third, CNN2Gate proposes an FPGA fitter algorithm that is based on reinforcement learning.

Using existing high-level synthesis technologies, it is possible to synthesize OpenCL Single Instruction Multiple Thread (SIMT) algorithms to RTL. It is worth mentioning some notable research efforts in that direction. In [20], the authors provided a deep learning accelerator targeting Intel's FPGA devices that are based on OpenCL. This architecture was capable of maximizing data-reuse and minimizing memory accesses. The authors of [21] presented a systematic design-space exploration methodology in order to maximize the throughput of an OpenCL-based FPGA accelerator for a given CNN model. They used synthesis results to empirically model the FPGA resource utilization. Similarly, in [22], the authors analyzed the throughput and memory bandwidth quantitatively in order to tackle the problem of design-space exploration of a CNN design targeting FPGAs. They also applied various optimization methods, such as loop-tiling, to reach the best performance. A CNN RTL compiler is proposed in [23]. This compiler automatically generates sets of scalable computing primitives to form an end-to-end CNN accelerator.

Another remarkable OpenCL-based implementation of CNNs is PipeCNN [8]. PipeCNN is mainly based on the capability of currently available design tools in order to use OpenCL kernels in high-level synthesis. PipeCNN consists of a set of configurable OpenCL kernels to accelerate CNN and optimize memory bandwidth utilization. Data reuse and task mapping techniques have also been used in that design. Recently, in [24], the authors added a sparse convolution scheme to PipeCNN in order to further improve its throughput. Our work (CNN2Gate) follows the spirit of PipeCNN. CNN2Gate is built on top of a modified version of PipeCNN. CNN2Gate is capable of exploiting a library of primitive kernels needed to perform inference of a CNN. In addition, CNN2Gate also includes means to perform automated design-space exploration and can automatically translate CNN models that are provided by a wide variety of machine learning libraries. It should be noted that our research goal is introducing a methodology to design an end-to-end framework to implement CNN models on FPGA targets. This means, without a loss of generality, that CNN2Gate can be modified to support other hardware implementations or technologies ranging from RTL to high-level synthesis with two conditions. First, CNN layers can be expressed as pre-defined templates. Second, the amount of parallelism in the hardware templates can be controlled.

There are several reasons that we have chosen PipeCNN as the foundation of CNN2Gate. First, while using the OpenCL model, it is possible to fine-tune the amount of parallelism in the algorithm, as explained in the Section 4.3. Second, it supports the possibility of having deep pipelined design, as explained in Section 3.2.2. Third, the library of primitive kernels can be easily adapted and re-configured based on the information that was extracted from a CNN model (Section 4.1).

Lately, reinforcement learning has been used in order to perform automated quantization for neural networks. HAQ [25] and ReLeQ [26] are examples of research efforts exploiting this technique. ReLeQ proposes a systematic approach for solving the problem of deep quantization automatically. This provides a general solution for the quantization of a large variety of neural networks. Likewise, HAQ, suggests a hardware-aware automated quantization method for deep neural networks while using actor-critic reinforcement learning method [27]. Inspired by these two papers, we used a reinforcement learning algorithm to control the level of parallelism in CNN2Gate OpenCL kernels.

Finally, it is worth mentioning the challenges of implementing digital vision system in hardware. In [28], the authors proposed a hardware implementation of a haze removal method exploiting adaptive filtering. Additionally, in [29], the authors provided an FPGA implementation of a novel method to recover clear images from degraded ones on FPGA.

3. Background

This section provides the background knowledge that is required to understand CNNs and OpenCL-based high-level synthesis workflows.

3.1. Convolutional Neural Networks (CNNs)

CNNs are categorized as feedforward networks. Figure 2 shows the most common architecture of a CNN. A CNN consists of one or several convolutional, pooling, and fully connected layers. These layers can be connected together to shape a deep convolutional neural network model. The data enters from the input, is processed by various hidden layers, and the classification results are presented as outputs.

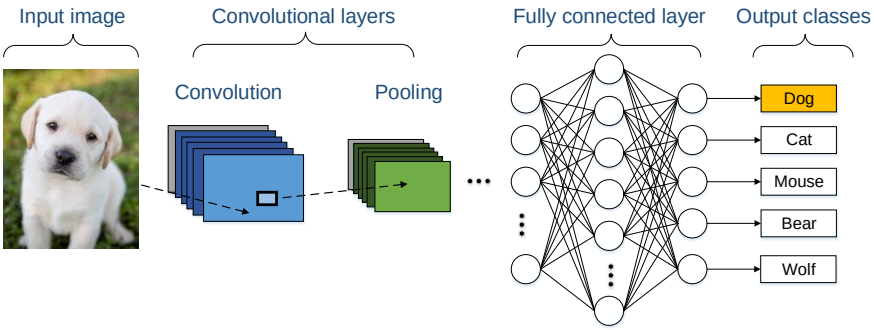

Figure 2. Demonstration of a Convolutional Neural Network (CNN) comprising an input image, convolution and pooling layers, fully connected layer, and output classification.

3.1.1. Convolutional Layers

Convolutional layers are used to extract features from the input data. A convolutional layer convolves the input with a convolutional kernel (filter) and passes the results to the next layer through a non-linear activation function. More specifically, each neuron in convolutional layers has a receptive field, and it is connected to other neurons in the adjacent layers through some learned weights and biases. The following equation shows that the feature F_k can be computed as:

$$F_k = f\left(W_k * I + b_k\right) \tag{1}$$

in which I is the input from the preceding layer or input image and W_k is the convolution kernel for feature k and b_k is the bias vector. The non-linear activation function is denoted $f(.)$ in (1).

3.1.2. Pooling Layers

The pooling layers are used in CNNs to down-sample and reduce the spatial resolution of the extracted features. The reduction of spatial resolution can help a CNN model to overcome input distortion and angular transformations. Pooling layers are usually made of max-pooling kernels. A max-pooling kernel selects the maximum value of the region of interest. The max-pooling feature can be defined as:

$$MPF_k = \max_{i,j \in \mathfrak{R}_{\mathrm{m,n}}} x_{k,(i,j)} \tag{2}$$

where MPF_k denotes k'th max-pooling feature and $\mathfrak{R}_{\mathrm{m,n}}$ shows the region of interest around point (m, n).

3.1.3. Fully Connected Layers

After extracting features of the input data by convolutional and pooling layers, the data are sent to a fully connected neural network that implements the decision making process. The fully connected layer interprets the extracted features and turns them into information represented with quantities. A softmax function is often used at the output of a fully connected layer in order to classify the output.

For more information on CNNs, readers are encouraged to read papers, such as [1,30].

3.2. OpenCL-Based High-Level Synthesis of CNNs

3.2.1. OpenCL High-Level Synthesis on FPGAs

OpenCL can be used in order to write programs that can be executed across heterogeneous platforms. OpenCL offers the ability to describe a parallel algorithm to be implemented on FPGA, for example.

However, this parallel description is at a level of abstraction that is much higher than hardware description languages, such as VHDL. An OpenCL application consists of two parts. The OpenCL "host" program that is written purely in C or C++ and that can be executed on any type of processor. On the other hand, "kernels" are accelerated functions that are implemented on some co-processor "device", such as an FPGA, while using fine-grained parallelism. The host can offload computation workload to kernels using sets of command queues. Figure 3 demonstrates this concept.

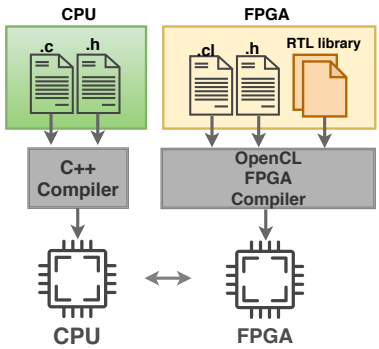

Figure 3. OpenCL high-level synthesis for Field Programmable Gate Arrays (FPGA).

3.2.2. OpenCL Pipes

In the OpenCL model, every kernel has access to the global memory of the device, as depicted in Figure 4a. In the case of massively parallel algorithms, memory transactions can be a bottleneck. In OpenCL 2.0 [13], "Pipes" were introduced to enable kernel-to-kernel communications. These tiny communication channels between kernels help to reduce the number of times kernels, refer to memory, and increase the memory access efficiency. In the case of FPGAs, these channels are implemented using FIFOs. Figure 4b shows that it is possible to stack OpenCL kernels on top of each other and pass data from one kernel to another. This feature makes FPGAs very well suited to implement stacked layers of CNNs as deeply pipelined kernels.

3.2.3. OpenCL High-Level Synthesis of CNNs for FPGAs

Leveraging the view from Figure 4b, a CNN model can be synthesized on FPGA while using deeply pipelined kernels as shown in Figure 4c. For this architecture, the following hardware accelerator kernels are needed: (1) Memory read (2) Memory write (3) Convolution (4) Pooling and (5) Fully connected. In many cases, convolution kernels and fully connected kernels can be fused together as a single 3-D matrix-matrix multiplication unit. Memory read/write kernels provide and store data for other kernels. This architecture has two main advantages: (1) Depending on the number of computing units in the convolution and pooling layer and the size of data fetched by memory read/write kernels, this architecture can be scalable (details will be discussed in the design-space exploration section) and (2) pipelined kernels can process data without storing the data moved between layers; this can significantly improve the memory access efficiency.

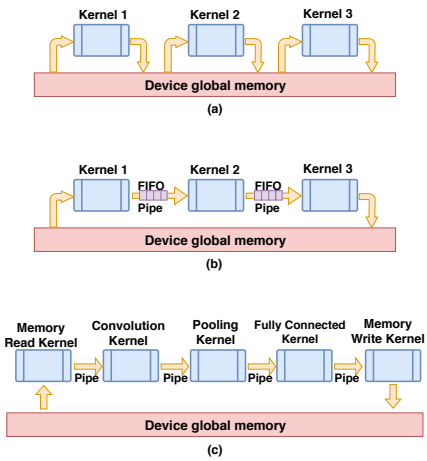

Figure 4. (**a**) Memory access pattern in OpenCL standard 1.x. (**b**) OpenCL pipes; in FPGAs, pipes are implemented as FIFOs. (**c**) Deeply pipelined CNN network architecture.

4. Proposed Architecture

4.1. Generalized Model Analysis Using ONNX

ONNX is a library that makes deep learning models interoperable. ONNX represents the computation flow of a deep neural network as an extensible computation graph model while using its own built-in operators and data-types. This inter-operable ecosystem eliminates the limitations that may stem from using any specific deep learning development framework. This makes it easier for hardware developers to exploit the most suited tool without being bound to the library with which the model was developed. ONNX provides the definition of a deep neural model while using extensible acyclic graphs. Nodes represent an operator and they have sets of inputs and outputs. ONNX can support a vast variety of tools, such as Pytorch, TensorFlow, Caffe, Keras, and more [18]. CNN2Gate offers a front-end ONNX parser for CNNs. Using ONNX as a transport layer decouples the high-level synthesis tool from the machine learning tool, as shown in Figure 1. CNN2Gate parses the computation dataflow—or the arrangement of layers—besides weights and biases for each layer. TheCN N2Gate parser traverses the ONNX graph nodes and extracts the synthesis information of each node based on the following operator types:

- Convolution: for the convolution operator, CNN2Gate parses dilations, pads, kernel shape, and stride. The reader can refer to [31] for more information regarding these variables and how they affect the computation. It also extracts the learned weights and biases for convolutional kernels. CNN2Gate also computes the output tensor size of the layer while using Equation (3). Let us assume the input of a two dimensional convolutional kernel is of size (c_{in}, h_{in}, w_{in}), where c denotes the number of features, h denotes the height, and w denotes the width. The output tensor size $(c_{out}, h_{out}, w_{out})$ can be written as:

$$\begin{aligned} h_{out} &= \left\lfloor \frac{h_{in} + 2p[0] - d[0](ks[0] - 1) - 1}{st[0]} + 1 \right\rfloor \\ w_{out} &= \left\lfloor \frac{w_{in} + 2p[1] - d[1](ks[1] - 1) - 1}{st[1]} + 1 \right\rfloor \end{aligned} \tag{3}$$

$$c_{out} = c_{in} \tag{4}$$

where ks is the kernel size, st is the stride, while p and d are the padding and dilation parameters, respectively.

- Max-pooling: similar to the convolution, CNN2Gate parses dilations, pads, kernel size, and strides. However, as max-pooling is a down-sampling kernel, it does not have weights and biases. The output tensor size of a max-pooling node with input size of (c_{in}, h_{in}, w_{in}) is identical to Equations (3) and (4).
- ReLu: CNN2Gate detects the presence of activation function such as "Relu" after a convolutional or max-pooling layer.
- General Matrix Multiplication (*"GEMM"*): a fully connected layer appears as a GEMM operator in ONNX dataflow graph. In CNN2Gate, there is no specific kernel for the fully connected layer.
- Softmax: CNN2Gate detects the presence of the softmax operator after a fully connected layer.

The front-end parser saves the information that specifies each layer's data in a linked structure in order to preserve the order. Later, this data structure is used by a high-level hardware synthesis tool. The preserved order serves as a guideline for the synthesizer to configure hardware pipelines.

4.2. Automated High-Level Synthesis Tool

Inspired from the Gajski-Kuhn chart [32], Figure 5 sketches the basic idea behind the CNN2Gate automated synthesis workflow. According to this diagram, the design of CNN2Gate is projected in three domains. The Algorithmic domain axis depicts the definition of concurrent algorithms and the Structural domain axis shows the building blocks to realize the Algorithmic domain. Finally, the Physical domain corresponds to the actual implementations in RTL. The lines connecting the dots show the relations between these domains. In the "Algorithmic Domain", CNN2gate parses the information from an ONNX model, as explained in Section 4.1. In the "Structural Domain", CNN2Gate uses 8-bit fixed point arithmetic units to perform computations. In addition, it configures memories, pipes, and kernels that correspond to the information that is received from the ONNX model. In the "Physical Domain", if weights and biases are floating point numbers, CNN2Gate can quantize these values based on the information that the user provides from post-training quantization [3]. In order to clarify further, CNN2Gate does not perform quantization itself; however, it can apply a given value that the user provides for a layer. This value can be expressed as an (N, m) pair, where the fixed-point weights/biases values are represented as $N \times 2^{-m}$. Moreover, CNN2Gate performs design-space exploration and generates Register Transfer Level (RTL) models targeting FPGAs.

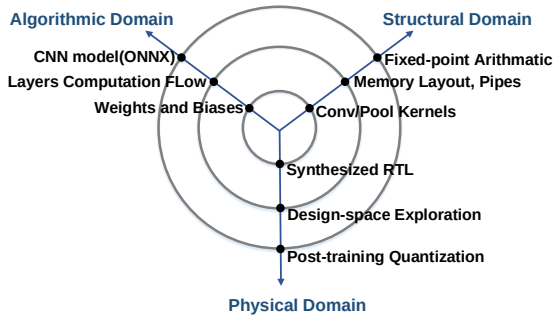

Figure 5. Gajski–Kuhn chart illustrating different aspects of CNN2Gate.

Note that CNN2Gate can automatically configure several memory buffers, depending on the layer operation type. For instance, if the next layer in the neural network is fully connected layer, it writes the data to the memory buffer that is associated with the fully connected layers and, similarly, if the layer is convolutional, it writes the data to the convolution buffer.

CNN2Gate is also capable of building and running the CNN model in both emulation and full flow mode. For the emulation mode, CNN2Gate compiles the project for a CPU. In some cases, the user needs to verify whether the CNN model performs correctly in terms of computational accuracy before committing to long hours of synthesis. The compilation for the emulation mode is significantly

faster—in the order of seconds—as compared to synthesizing the full flow for the FPGA, which takes several hours. This feature makes the workflow more versatile for the designers who want to iterate between a FPGA design and a CNN model to reach the best quantization parameters and accuracy. In order to synthesize the full flow on FPGA, CNN2Gate accepts the name of the FPGA board to perform design-space exploration (Section 4.3) and generates the RTL accordingly. In order to validate CNN2Gate, we tested this process by targeting three different Intel™ FPGA boards [33–35] and report the results later in this paper. In addition, we used Intel OpenCL SDK 16.1 to synthesize our designs.

Taking advantage of OpenCL flexibility, it is possible to design an architecture with several degrees of freedom. The first degree of freedom is the size of the pipes. The better throughput of the pipes means less congestion point for data to be moved from a kernel to another. The second degree of freedom is the bandwidth of data that memory write/read kernels provide. The third degree of freedom is the number of parallel convolutional (CONV) and RELU units that are used to implement convolution kernels. Figure 6 shows this concept in a simple example. These degrees of freedom for deeply pipelined kernels are leveraged from what proposed by [8]. The memory read kernel fetches N_l vectors of size N_i for features and weights. Tuning N_i and N_l can provide a better throughput for data write/read kernels. Note that the memory access schedule of where and when to read the features and weights are derived by the host program. The memory access schedule is configured by the front-end parser that is based on the CNN model . The number of computation lanes (N_l) shows the level of parallelism of the algorithm. The number of CONVs in a convolution kernel, the size of data pipes and the number of max-pool operators in the max-pool kernel are tuned according to N_l. Changing N_l and N_i can result in different utilization ratios of FPGA resources. For instance, in the case of increasing N_l, (1) more on-chip memory in read/write kernels, (2) more register for pipe FIFOs, (3) more DSP slices for CONVs, and (4) more LUT for max-pooling kernels are needed in order to accommodate the design on FPGA.

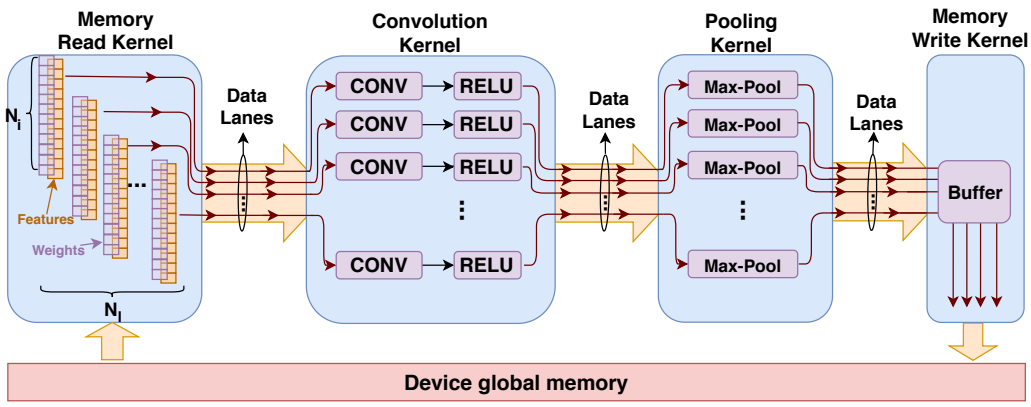

Figure 6. Detailed demonstration of vectorized input data and weights and the concept of computation lanes in pipelined kernels.

Architectural Limitations

The architecture shown in Figure 6 is used to perform the calculation of all layers. In order to obtain practical implementations and allow the targeting FPGAs of various size, it is necessary to fold (or time multiplex) the layers onto fewer hardware lanes and vectors [9,10]. Therefore, arbitrary choices for N_l and N_i are not always possible. N_i should be a divisor of the features' width for all layers in order to avoid padding. Likewise, N_l should be a divisor of the number of features for all layers to avoid idle lanes in some layers.

4.3. Hardware-Aware Design-Space Exploration

CNN2Gate analyzes the computation flow layers that are extracted from the model by the ONNX front-end parser and determines all options possible for N_l and N_i. CNN2Gate is also capable of interacting with the Intel OpenCL compiler to estimate resource usage for a specific choice of N_l and N_i. Thus, there is a need to identify the best values for N_l and N_i. The process of finding the best values of N_l and N_i is what we call design space exploration. Many methods can be considered to do it and three such methods are investigated in this section.

4.3.1. Brute Force Design Space Exploration (BF-DSE)

This method exhaustively searches for all possible pairs of N_l and N_i, and it finds the feasible option that maximizes FPGA resource utilization. In our case, the solution maximizing resource utilization corresponds to the one providing the best throughput. This method is simple to execute and it always find the best solutions. However, for larger FPGAs with large number of possible candidates for N_l and N_i, an exhaustive brute force search could take excessive time. In the next section, a more efficientsearch strategy is described.

4.3.2. Reinforcement Learning Design Space Exploration (RL-DSE)

RL-DSE trains a reinforcement learning (RL) agent to find the best options for N_l and N_i. Reinforcement learning for design-space exploration is of interest for two reasons. First, it can be faster than brute force and second, it can be merged with other RL-agents, such as HAQ [25] or ReleQ [26] to determine the level of parallelism and the quantization of each layer. RL-DSE explores hardware options for N_l and N_i, and finds the best fit for the specified hardware. The RL-DSE agent receives a reward signal that corresponds to the feedback provided by the Intel OpenCL compiler for FPGA resource utilization. Hence, the reward function for RL-DSE is designed in order to maximize FPGA resource utilization.

When CNN2Gate triggers Intel OpenCL compiler to evaluate a hardware option, it receives the corresponding hardware resource utilization. This feedback comprises (1) the percentage of lookup table utilization, (2) the percentage of DSP utilization, (3) the percentage of on-chip memory block utilization, and (4) the percentage of register utilization. We denote these percentages P_{lut}, P_{dsp}, P_{mem}, and P_{reg}, respectively.

Given the percentages of resource utilization, the agent takes a series of actions. The agent starts from the minimum values of N_l and N_i. RL-DSE can flexibly choose to (1) increase N_l, (2) increase N_i, or (3) increase both N_i and N_l. If one of the variables reaches the maximum possible value that is based on the CNN topology, the variable is reset to its initial value.

Let us assume that the average usage factor is defined as:

$$F_{avg} = \frac{P_{lut} + P_{dsp} + P_{mem} + P_{reg}}{4} \tag{5}$$

Further assume that the user defines a vector of thresholds T_{th} for the maximum usage that is tolerated for each quota. The reward function can be described, as follows.

In Algorithm 1, H_{best} is the best hardware options and F_{max} is the maximum usage that is observed by the agent during the exploration of the search environment. In the reward shaping function, if at least one of the hardware utilization quotas (P_{lut}, P_{dsp}, P_{mem}, P_{reg}) exceeds the thresholds specified in the vector of threshold limits ($T_{th} = (T_{lut}, T_{dsp}, T_{mem}, T_{reg})$), the agent receives a negative reward for this choice and the chance of choosing this option for a later iteration decreases. The reward function keeps track of the maximum usage score F_{max}, and constantly update F_{max} and H_{best} with the best hardware option observed by the agent in the environment. Becasue, during exploration, the best value of the reward function is unknown, besides exhaustion of the search space, there is no exact stop condition for agent exploration. In this case, we used a variation of time-limited reinforcement

learning [36] with which, the number of iterations in each episode is limited. Finally, in our RL-DSE, a scaling factor $\beta = 0.01$ is applied to F_{avg} in order to form the final reward function to convert it from percentage scale to a number between 0 and 1.

Note that a discount factor $\gamma = 0.1$ is used in our RL agent design. The discount factor specifies that the agent does not have unlimited time to find the best option. Thus, the agent receives a down-scaled reward as it spends more time in the environment. The discount factor urges the agent to optimize the total discounted reward [37]:

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+1} + ... = \sum_{i=1}^{n} \gamma^i r_{t+i} \tag{6}$$

where r_t is the reward calculated in time t according to Algorithm 1.

Algorithm 1: Reward shaping

```
Input  : T_th , F_avg, P_lut, P_dsp, P_mem, P_reg, N_i and N_l
Output : Reward , H_best
Variables: F_max, T_th = (T_lut, T_dsp, T_mem, T_reg)
if (P_lut, P_dsp, P_mem, P_reg) < (T_lut, T_dsp, T_mem, T_reg) then
    if F_avg > F_max then
        F_max = F_avg
        Reward = β × F_avg
        H_best = (N_i, N_l)
    else
        Reward = 0
    end
else
    Reward = −1
end
```

4.3.3. Hill-Climbing Design Space Exploration (HC-DSE)

While exhaustive enumeration of all possible hardware options (BF-DSE) is very efficient in small design-spaces, the execution time increases proportionally to the number of possibilities in the design-space. There are other smart optimization algorithms that find the best solution significantly faster than brute-force. Hill-climbing is an iterative numerical algorithm that starts from an arbitrary location in the design space and attempts to find a better solution by moving incrementally in order to optimize the target function. As shown in Figure 7, optimizer O examines all possible successors (in the 2D case A, B and C) and calculates the relative changes to the target function corresponding to those successors (Δ_A, Δ_B and Δ_C). Afterward, the optimizer moves toward the best choice. This process continues until there is no further improvement in the target function or, in our case, it continues until the design does not fit on the target FPGA. This simple algorithm is very efficient in finding local optimum. Algorithm 2 shows the pseudo code for hill-climbing in our context.

Note that the hill-climbing algorithm is guaranteed to find the best solution in a **convex** design-space.

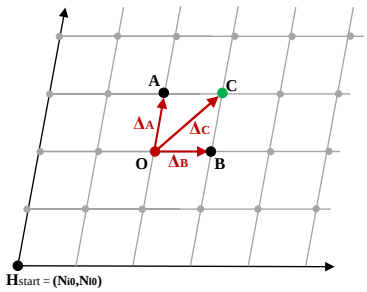

Figure 7. The hill-climber in a two-dimensional (2D) design-space. Optimizer O moves toward the direction that optimizes the cost function Δ.

Algorithm 2: Hill-climbing

```
Input  : H_start = (N_i0, N_l0)
Output : H_best
H_current := H_start
compute: F_avg(current)
while true do
    H_next := null
    F_avg(next) := −inf
    for i : Neighbours(H_current) do
        if (P_lut, P_dsp, P_mem, P_reg)_i < (T_lut, T_dsp, T_mem, T_reg) then
            if F_avg(i) > F_avg(next) then
                F_avg(next) = F_avg(i)
                H_next = H_i
            end
        end
    end
    if F_avg(next) ⩽ F_avg(current) then
        H_best = H_current
        return : H_best
    end
end
```

5. Results

Table 1 shows the execution times of AlexNet [38] and VGG-16 [39] for three platforms while using CNN2Gate. The user can verify the CNN model on CPU using the CNN2Gate emulation mode in order to confirm the resulting numerical accuracy, as mentioned before. Even if the execution time is rather large, this is a very useful feature to let the developer verify the validity of the CNN design on the target hardware before going forward for synthesis, which is a very time-consuming process. Note that the emulation's execution time cannot be a reference for the throughput performance of a core-i7 processor. The emulation mode only serves the purpose of verifying the OpenCL kernels operations. In [40], the authors described the execution of AlexNet on desktop CPUs and reported an execution time as low as 2.15 s. The reported results also show the scalability of this design Indeed, the results are reported for both the low cost Cyclone V SoC and the much more expensive Arria 10 FPGA that has much more resources. The exploited level of parallelism was automatically extended by the design-space exploration algorithm in order to obtain better results for execution times that are commensurate with the capacity of the hardware platform.

Table 1. Execution times for Alexnet and VGG (batch size = 1).

Platform	Resource Type	Execution Time		f_{max}
		AlexNet	VGG-16	
Core-i7 (Emulation)	CPU	13 s	148 s	N/A
Cyclone V 5CSEMA5	System-on-Chip	153 ms	4.26 s	131 MHz
Arria 10 GX 1150	CPU+FPGA with PCIe link	18 ms	205 ms	199 MHz

In order to maintain the scalability of the algorithm, it is not always possible to use arbitrary choices for (N_i, N_l) parameters. These parameters must be chosen in a way that kernels can be used as the building block of all the layers. This leads to have limited options to increase the level of parallelism that can be exploited with a given network algorithm. Relaxing this limitation (i.e., manually designing kernels based on each layers' computation flow) could lead to better resource consumption and higher throughput at the expense of losing the scalability that is needed for automation of this process. The maximum operating frequency (f_{max}) varies for different FPGAs. Indeed, it depends on the underlying technology and FPGA family. Intel OpenCL compiler (synthesizer) automatically adjust PLLs on the board to use the maximum frequency for the kernels. It is of interest that the operating frequency of the kernels supporting AlexNet and VGG-16 were the same, as they had essentially the same critical path, even though VGG-16 is a lot more complex. The larger complexity of VGG was handled by synthesizing a core (i.e., Figure 6) that executes a greater number of cycles if the number of layers in the network increases.

Table 2 gives more details regarding the design-space exploration algorithms which are coupled with synthesis tool. All three algorithms use the resource utilization estimation of the synthesizer to fit the design on the FPGA. This is important as the time consumed for design-space exploration is normally under 5 min., while the synthesis time for larger FPGAs, such as the Arria 10, can be close to 10 h.

Table 2. CNN2Gate Synthesis and Design-Space Exploration Details (AlexNet).

Platform	HC-DSE Time	RL-DSE Time	BF-DSE Time	Synthesis Time	Resources Available	Resources Consumed	Hardware Options (N_i, N_l)
Cyclone V SoC 5CSEMA4	20 s	2.5 min	3.5 min	N/A	**ALM:** 15 K **DSP:** 83 **RAM blocks:** 321	Does not fit	N/A
Cyclone V SoC 5CSEMA5	1 min	2.5 min	3.5 min	46 min	**ALM:** 32 K **DSP:** 87 **RAM blocks:** 397 **Mem. bits:** 4 M	**ALM:** 26 K **DSP:** 72 **RAM blocks:** 397 **Mem. bits:** 2 M	(8, 8)
Arria 10 GX 1150	2 min	3 min	4 min	8.5 hrs	**ALM:** 427 K **DSP:** 1516 **RAM blocks:** 2713 **Mem. bits:** 55.5 M	**ALM:** 129 K **DSP:** 300 **RAM blocks:** 1091 **Mem. bits:** 16 M	(16, 32)

Experimenting with various DSE algorithms confirms that smart algorithms (such as reinforcement learning and hill-climbing) can provide advantages for design-space exploration when compared to brute-force enumeration. The first goal of suggesting the RL-DSE and HC-DSE methods is to demonstrate that it is possible to further decrease the exploration time by using some better search methods. Analyzing the execution times shows that the reinforcement learning algorithm is almost 25 percent and the hill-climbing is 50 percent faster than the brute-force algorithm when optimizing the

design for the Arria 10 FPGA. Note that these exploration times are significantly less than the synthesis time, since we use estimation of resource usage provided by the synthesizer. On the other hand, performing full synthesis as part of the exploration process is not advisable, because the exploration time would take weeks. Because HC-DSE follows the gradient of resource usage, it is always faster than BF-DSE. This method is the best when the design-space is convex. However, if the search space is not convex (e.g., having several local maximums), HC-DSE might choose a wrong solution. In contrast, it is less probable for RL-DSE to be trapped in a local maximum due to the random nature of the reinforcement learning algorithm. Moreover, the RL-DSE algorithm would be more valuable if it could be exploited in conjunction to the reinforcement learning quantization algorithms, such as ReLeQ [26].

The goal of considering various DSE algorithms in our work is to provide a versatile tool for the user in different conditions and it is not limited to a specific case. We included the brute-force algorithm in CNN2gate in order to guarantee a successful exploration for small design-spaces. However, presuming that the design space is small is not always a correct assumption. For large convex design-spaces, we added the hill-climbing algorithm to find the best solution which works significantly faster than brute-force. For non-convex and large design-spaces, reinforcement learning was found to work well [41]. In Table 2, the design-space is small. This is why the difference between the execution time between various exploration algorithms is small. However, it was shown that HC-DSE can be twice faster than brute force when optimizing the design for the Arria 10 FPGA.

There are other model-based design-space exploration algorithms that are dedicated to a specific implementation of a library of primitives. For instance, in [24], the authors proposed a performance model for their design, and they can predict the performance of the hardware implementation based on the resource requirements. The advantage of our proposed DSE algorithms is that our algorithms are model agnostic. This means that the CNN2Gate framework tunes the parallelism parameters of the design and directly queries the performance feedback from the synthesizer.

We tried CNN2Gate on three platforms. The first one is a very small Cyclone V device with 15K adaptive logic modules (ALMs) and 83 DSPs. The fitter could not fit either ALexNet or VGG on this device. Clearly, the minimum space that is required for fitting this design on FPGA is fairly large due the complexity of the control logic. CNN2Gate did not experience any difficulty fitting the design on bigger FPGAs. as demonstrated in Table 2. Resource utilization and hardware options (N_i, N_l) are also provided, which correspond to the execution times that are shown in Table 1.

CNN2Gate resource consumption is very similar for AlexNet and VGG-16. In the case of identical hardware options, CNN2Gate's synthesized core is going to be nearly identical for all CNN architecture as shown in Figure 6. The only difference is the size of internal buffers to allocate the data in the computation flow. More quantitatively, in our implementation, VGG-16 uses 8% more of the Arria 10 FPGA block RAMs in comparison to what is shown in Table 2 for AlexNet.

Revisiting Figure 6, pipelined kernels are capable of reading data from global memory and processing the convolution and pooling kernel at the same time. In addition, for fully connected layers in CNNs, the convolution kernel acts as the main data process unit and the pooling kernel is configured as a pass-through. When considering this hardware configuration, we can merge convolution and pooling layers as one layer. In the case of AlexNet, this leads to five fused convolution/pooling and three fully-connected layers. Figure 8 reports the detailed execution time of these six layers, including memory read and write kernels for each layer. Thus, as the algorithm proceeds through the layers, the dimensions of the data (features) reduced and the execution time is decreased. Note that, in this figure, Layer 1 to Layer 5 are fused layers comprising a convolution, a pooling, and a ReLu layers. Additionally, note that, although the number of parameters in convolutional layers are less in fully connected layers, the memory consumption is far greater than with fully connected layer. Therefore, the performance of the first convolutional layers can be affected in a system accessing external memory (RAM).

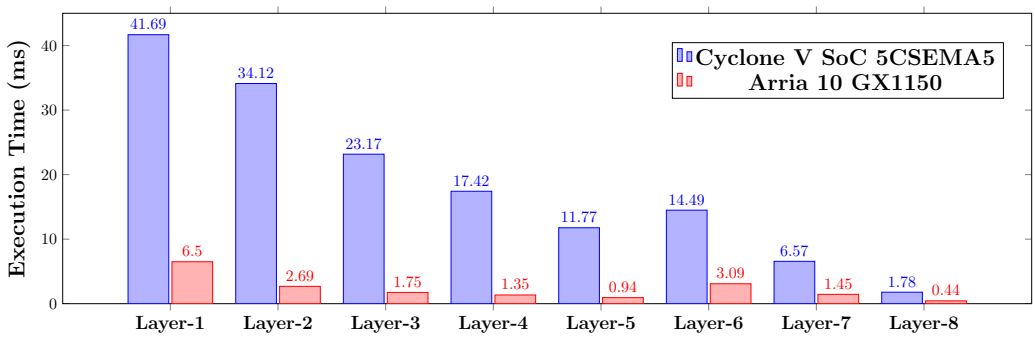

Figure 8. Detailed breakdown of execution time for each layer of AlexNet. A Layer here means execution of one round of pipelined kernels as shown in Figure 6. In case of AlexNet Layer-1 to Layer-5 are combination of memory read/write, convolution and pooling kernels, while Layer-5 to Layer-7 are combination of memory read/write and fully connected kernels.

Table 3 shows a detailed comparison of CNN2Gate to other existing works for AlexNet. CNN2Gate is faster than [21,22] in terms of latency and throughput. However, CNN2Gate uses more FPGA resources than [21]. To make a fair comparison, we can measure relative performance density as per DSP or ALMs. In this case, the CNN2Gate performance density (GOp/s/DSP) is higher (0.266) when compared to 0.234 for [21]. There are other designs, such as [9,23], which are faster than our design in terms of pure performance. Nevertheless, the CNN2Gate method that is outlined above significantly improves on existing methods, as it more scalable and automatable than the methods that are presented in [9,23], which are limited in these regards, as they require human intervention in order to reach high levels of performance.

Table 3. Comparison to the existing works AlexNet with hardware options $(N_i, N_l) = (16, 32)$

	AlexNet [22]	**AlexNet [23]**	**AlexNet [9]**	**AlexNet [21]**	**AlexNet [This Work] ***
FPGA	Virtex-7 VX485T	Stratix-V GXA7	Zynq 7045	Stratix-V GX-D8	Arria 10 GX1150
Synthesis method	C/C++	RTL	C/C++	OpenCL	OpenCL
Frequeny (MHz)	100	100	125	-	199
Logic Utilization	186K (61%)	121K (52%)	-	120K (17%)	129K (30%)
DSP Utilization	2240 (80%)	256 (100%)	897 (99.5%)	665 (34%)	300 (20%)
Latency (ms)	21.61	12.75	8.22	20.1	18.24
Precision (bits)	32 float	8–16 fixed	16 fixed	8–16 fixed	8 fixed
Performance (GOp/s)	61.62	114.5	161.98	72.4	80.04

* batch size = 1.

Second, the design entry of [9,23] are C and RTL, respectively, while our designs were automatically synthesized from ONNX while using OpenCL. Thus, not surprisingly, our work does not achieve the maximum reported performance. This is partly due to the use of ONNX as a starting point, and trying to keep the algorithm scalable for either large and small FPGAs. This imposes some limitations, such as the maximum number of utilized CONV units per layer. There are also other latency reports in the literature, such as [8]. However, those latency reports are measured with favorable batch size (e.g., 16). An increasing batch size can make more parallelism available to the algorithm that can lead to higher throughput. Thus, for clarity, we limited the comparisons in Table 3 and 4 to batch size = 1.

Table 4. Comparison to the existing works VGG-16 with hardware options $(N_i, N_l) = (16, 32)$.

	VGG-16 [42]	**VGG-16 [11]**	**VGG-16 [9]**	**VGG-16 [21]**	**VGG-16 [This Work] ***
FPGA	Zynq 7045	Arria 10 GX1150	Zynq 7045	Stratix-V GX-D8	Arria 10 GX1150
Synthesis method	-	RTL	C/C++	OpenCL	OpenCL
Frequeny (MHz)	150	150	125	120	199
Logic Utilization	182 (83.5%)	161K (38%)	-	-	129K (30%)
DSP Utilization	780 (89.2%)	1518 (100%)	855 (95%)	-	300 (20%)
Latency (ms)	-	47.97	249.5	262.9	205
Precision (bits)	16 fixed	8–16 fixed	16 fixed	8–16 fixed	8 fixed
Performance (GOp/s)	136.91	645.25	161.98	117.8	151.7

* batch size = 1.

Table 4 shows a detailed comparison of CNN2Gate to other existing works for VGG-16. It is observable that CNN2Gate is performing better for larger neural networks, such as VGG. CNN2Gate achieves 18 % lower latency than [9], despite the fact that CNN2Gate uses fewer DSPs. While, for AlexNet, [9] was more than 50 % faster than CNN2Gate. Finally, for VGG-16, we did find some hand tailored RTL custom designs, such as [11], which are faster than CNN2Gate.

6. Discussions and Generalization

In this section, we explore CNN2Gate from the point of view of its software architecture. We also discuss some limitations of CNN2Gate that is caused by the library of primitives over which it is built. Finally, generalized forms of design-space search algorithms are discussed, which can be very insightful in future designs.

6.1. CNN2Gate as an External Controller with a Synthesizer in the Loop

It is helpful to see this framework in the context of a controller in order to better understand the design of CNN2Gate. CNN2Gate provides two sets of actions as a controller:

1. Arranging the basic hardware building blocks or library of primitives: CNN2Gate provides the framework in which the primitive blocks are connected to form the specific architecture of a convolutional neural network. This architecture is based on the model that is parsed from an ONNX representation of the neural network. As shown in Figure 9, CNN2Gate chooses and connects the building blocks of a convolutional neural network from a library of primitives. This library of primitives can be designed by the user or it can be adapted from other existing open-source designs. We assume that the level of parallelism is controllable in the library of primitives.
2. Providing parallelism control for algorithm or design-space exploration: after providing the layout of building blocks, CNN2Gate makes several queries to the hardware synthesizer and determines the best parameters particularly regarding the level of parallelism in order to obtain the best performance or performance/accuracy/cost trade-off of the hardware implementation. The performance characteristics of the hardware architecture can be defined by the user as the total logic utilization, throughput, or combination of them.

In our case, we adapted the library of primitives from PipeCNN [8]. The three DSE algorithm that we are introduced in the previous sections are the control algorithms that work in conjunction with the synthesizer in order to fit the design to the desired FPGA target.

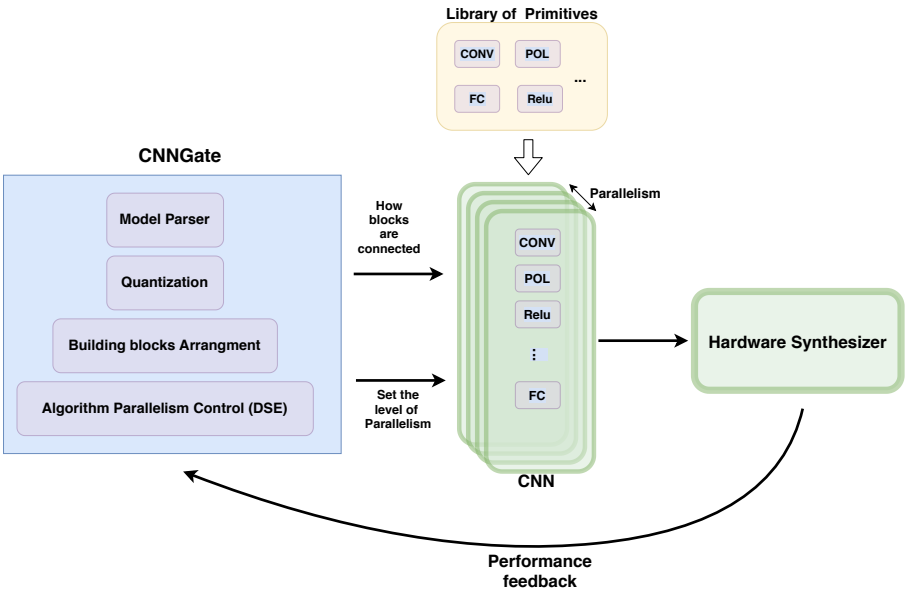

Figure 9. CNN2Gate as an external controller.

6.2. Ablation Study of the Basic Hardware Building Blocks

Because CNN2Gate's primitive libraries are adapted from [8], it inherits the limitations of PipeCNN. Nevertheless, the CNN2Gate is flexible and can be easily adapted to other primitive libraries. Being in the form of an external controller, CNN2Gate offers the possibility of continuous integration to new DSE algorithms and primitive hardware libraries. For instance, a more recent version of the PipeCNN primitive library were introduced to support RESNET [43]. In this case, the CNN2Gate parser block can be changed to support the new architecture. Moreover, the design-space algorithms can stay the same.

The methods that CNN2Gate uses are flexible enough to be adapted for the implementation of Recurrent Neural Networks (RNNs). This requires the user to provide the primitive libraries of the RNN. The number of parameters that control the parallelism in the algorithm might change when the library of primitive changes completely. This requires having a generic N-dimensional design-space exploration. In the next Section, possible generalizations of design-space algorithms are explained.

6.3. Generalizations of Design-Space Exploration Algorithms

In Section 4.3, some algorithms are introduced in order to explore the two-dimensional design space. These algorithms can be easily generalized to N-dimensional explorations. The following sections demonstrate this generalization.

6.3.1. N-Dimensional Hill-Climbing Design Space Exploration

The hill-climbing algorithm is widely used in artificial intelligence to find local maximums. The relative simplicity of this algorithm makes it the first choice for our exploration algorithms. The hill-climbing algorithm will find the global maximum if the search space is convex.

A hill-climbing optimizer starts from an initial point $H_0 = (x_{10}, x_{20}, ..., x_{n0})$, as shown in Algorithm 3. This initial point can be chosen randomly in the search space. In each step, the optimizer visits all the neighbours of the current location and examine the objective function O, if the optimizer finds a better choice among the neighbours , it updates its position to the better choice. This procedure continues iteratively until the hill climber reaches the local maximum and cannot find a better point in its neighborhood. Note that the objective function O can be defined as the total logic utilization, throughput, or combination of them.

Algorithm 3: Generalized Hill-climbing Algorithm

```
Input  : H_0 = (x_10, x_20 ... x_n0)
Output : H_best
H_current := H_start
compute: O(H_current)
while true do
    H_next := null
    O(H_next) := −inf
    for i : Neighbours(H_current) do
        if (H_i satisfies boundary limitations then
            if O(H_i) > O(H_next) then
                O(H_next) = O(H_i)
                H_next = H_i
            end
        end
    end
    if O(H_next) ⩽ O(H_current) then
        H_best = H_current
        return : H_best
    end
end
```

6.3.2. N-Dimensional Reinforcement Learning Design Space Exploration

A reinforcement learning (RL) software agent takes actions in the environment in order to maximize the reward, as shown in Figure 10. In the context of CNN2Gate, an RL agent is used in order to obtain the optimal control to choose the best hardware options for the model in the synthesis process. When considering CNN2Gate as an external controller (i.e., Figure 9) fits very well with the paradigm of a reinforcement learning agent. In this context, CNN2Gate is the *'Agent'* that explores the available synthesis options of the convolutional neural networks (i.e., *'Environment'*). Controlling the various characteristics of a neural network using reinforcement learning has recently appeared in the literature. For instance, in [25], an RL agent is used to find the best quantization level for each layer.

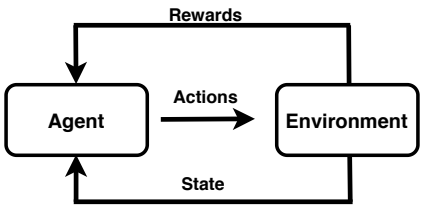

Figure 10. Reinforcement Learning: a software agent takes actions in the environment in order to maximize the reward.

To set up the RL agent to explore the N-dimensional design-space, we need to take into consideration the following steps:

1. Defining the states: in the context of a hardware optimizer, states can be defined as a set of all possible hardware options. Particularly, states represent the whole design-space.
2. Defining the actions: actions are the act of moving from one state to another. In the case of N-dimension, actions can be defined as the unit vector of each dimension.

3. Forming a reward function: executing an Action a in the state s provides the agent a numerical reward score r. The agent tries to maximize this reward. In the context of this article, the reward can be throughput.
4. Considering a living penalty or discount factor for the agent: the agent should not have an infinite amount of time to explore the environment. The longer the agent stays in the environment, the less reward it should get. This trains the agent to find the best hardware option as fast as possible.

Having defined all of the previous steps, we are able to train our agent to find the best hardware option while using common reinforcement learning techniques, such as Q-Learning [44].

7. Conclusions

This paper described the design and implementation of a general framework for developing convolutional neural networks on FPGAs. This framework takes the form of a Python library that can be integrated with a wide range of popular machine learning frameworks. CNN2Gate makes it easy for machine learning developers to program and use FPGAs in order to perform inference. CNN2Gate exploits the OpenCL synthesis workflow for FPGAs that are offered by commercial vendors. CNN2Gate is capable of parsing the data-flow of CNN models expressed in ONNX. This framework also has an integrated design-space exploration tool that helps developers to find the best hardware option for synthesizing a given CNN model on an FPGA. CNN2Gate achieves a classification latency of 205 ms for VGG-16 and 18 ms for AlexNet on an Intel Arria 10 FPGA. These results are excellent when considering that they were obtained by an automated design space exploration and synthesis process, which is not relying on expert's low-level hardware knowledge.

Author Contributions: All authors contributed substantially to methodology and validation of this manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by an IVADO (Institut de Valorisation des Données) grant through its fundamental research program and the work was conducted with tools provided by CMC Microsystems.

Acknowledgments: The authors would like to thank the Institut de Valorisation des Données (IVADO) and CMC Microsystems for supporting this research.

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

Abbreviations

The following abbreviations are used in this manuscript:

BF	Brute Force
CNN	Convolutional Neural Network
CNN2Gate	Convolutional Neural Network to field programmable gate arrays.
DSE	Design Space Exploration
FPGA	Field Programmable Gate Array
FIFO	First In First Out
HC	Hill Climbing
RL	Reinforcement Learning
ONNX	Open Neural Network Exchange

References

1. Rawat, W.; Wang, Z. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Comput.* **2017**, *29*, 2352–2449. [CrossRef] [PubMed]
2. Strigl, D.; Kofler, K.; Podlipnig, S. Performance and scalability of GPU-based convolutional neural networks. In Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy, 17–19 February 2010; pp. 317–324.

3. Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv* **2018**, arXiv:1806.08342.
4. Wang, N.; Choi, J.; Brand, D.; Chen, C.Y.; Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, QC, Canada, 3–8 December 2018; pp. 7675–7684.
5. Nurvitadhi, E.; Venkatesh, G.; Sim, J.; Marr, D.; Huang, R.; Ong Gee Hock, J.; Liew, Y.T.; Srivatsan, K.; Moss, D.; Subhaschandra, S.; et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 5–14.
6. Li, S.; Sun, K.; Luo, Y.; Yadav, N.; Choi, K. Novel CNN-Based AP2D-Net Accelerator: An Area and Power Efficient Solution for Real-Time Applications on Mobile FPGA. *Electronics* **2020**, *9*, 832. [CrossRef]
7. Intel. Intel User-Customizable Soc FPGAs. 2019. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01167-custom-arm-soc.pdf (accessed on 19 December 2020).
8. Wang, D.; Xu, K.; Jiang, D. PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In Proceedings of the 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, Australia, 11–13 December 2017; pp. 279–282.
9. Venieris, S.I.; Bouganis, C.S. fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs. *IEEE Trans. Neural Netw. Learn. Syst.* **2018**, *30*, 326–342, [CrossRef] [PubMed]
10. Umuroglu, Y.; Fraser, N.J.; Gambardella, G.; Blott, M.; Leong, P.; Jahre, M.; Vissers, K. Finn: A framework for fast, scalable binarized neural network inference. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 65–74.
11. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 45–54.
12. Bilaniuk, O.; Wagner, S.; Savaria, Y.; David, J.P. Bit-Slicing FPGA Accelerator for Quantized Neural Networks. In Proceedings of the 2019 IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 26–29 May 2019; pp. 1–5.
13. Vasiljevic, J.; Wittig, R.; Schumacher, P.; Fifield, J.; Vallina, F.M.; Styles, H.; Chow, P. OpenCL library of stream memory components targeting FPGAs. In Proceedings of the 2015 International Conference on Field Programmable Technology (FPT), Queenstown, New Zealand, 7–9 December 2015; pp. 104–111.
14. Duarte, J.; Han, S.; Harris, P.; Jindariani, S.; Kreinar, E.; Kreis, B.; Ngadiuba, J.; Pierini, M.; Rivera, R.; Tran, N.; et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* **2018**, *13*, P07027. [CrossRef]
15. Zhang, C.; Sun, G.; Fang, Z.; Zhou, P.; Pan, P.; Cong, J. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *38*, 2072–2085 [CrossRef]
16. Feist, T. Vivado design suite. *White Pap.* **2012**, *5*, 30.
17. Intel. Intel Quartus Prime Software. Available online: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html (accessed on 19 December 2020).
18. ONNX. Open Neural Network Exchange Format. Available online: https://onnx.ai/ (accessed on 19 December 2020).
19. hls4ml Project Current Status. Available online: https://hls-fpga-machine-learning.github.io/hls4ml/STATUS.html (accessed on 13 July 2019).
20. 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; pp. 55–64.
21. 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; pp. 16–25.

22. 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; pp. 161–170.
23. Ma, Y.; Suda, N.; Cao, Y.; Seo, J.S.; Vrudhula, S. Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; pp. 1–8.
24. Wang, D.; Xu, K.; Jia, Q.; Ghiasi, S. ABM-SpConv: A Novel Approach to FPGA-Based Acceleration of Convolutional Neural Network Inference. In Proceedings of the 56th Annual Design Automation Conference 2019, Las Vegas, NV, USA, 2–6 June 2019; p. 87.
25. Wang, K.; Liu, Z.; Lin, Y.; Lin, J.; Han, S. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019; pp. 8612–8620.
26. Yazdanbakhsh, A.; Elthakeb, A.T.; Pilligundla, P.; Esmaeilzadeh, F.M.H. ReLeQ: An Automatic Reinforcement Learning Approach for Deep Quantization of Neural Networks. *arXiv* **2018**, arXiv:1811.01704.
27. Grondman, I.; Busoniu, L.; Lopes, G.A.; Babuska, R. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.)* **2012**, *42*, 1291–1307. [CrossRef]
28. Zhang, B.; Wei, J. Hardware Implementation for Haze Removal With Adaptive Filtering. *IEEE Access* **2019**, *7*, 142498–142506. [CrossRef]
29. Ngo, D.; Lee, S.; Lee, G.D.; Kang, B. Single-Image Visibility Restoration: A Machine Learning Approach and Its 4K-Capable Hardware Accelerator. *Sensors* **2020**, *20*, 5795. [CrossRef] [PubMed]
30. Véstias, M.P. A survey of convolutional neural networks on edge with reconfigurable computing. *Algorithms* **2019**, *12*, 154. [CrossRef]
31. Dumoulin, V.; Visin, F. A guide to convolution arithmetic for deep learning. *arXiv* **2016**, arXiv:1603.07285.
32. Gajski, D.D.; Kuhn, R.H. New VLSI tools. *Computer* **1983**, 11–14. [CrossRef]
33. Terasic. DE0-Nano-SoC Kit/Atlas-SoC Kit. 2019. Available online: de0-nano-soc.terasic.com (accessed on 19 December 2020).
34. Terasic. DE1-SoC Board. 2019. Available online: de1-soc.terasic.com (accessed on 19 December 2020).
35. Nallatech. Nallatech 510 Acceleration Board. 2019. Available online: https://www.bittware.com/fpga/510t/ (accessed on 19 December 2020).
36. Mnih, V.; Badia, A.P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016; pp. 1928–1937.
37. Van Hasselt, H.; Wiering, M.A. Reinforcement learning in continuous action spaces. In Proceedings of the 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, Honolulu, HI, USA, 1–4 April 2007; pp. 272–279.
38. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*; ACM: New York, NY, USA, 2012; pp. 1097–1105. [CrossRef]
39. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv* **2014**, arXiv:1409.1556.
40. Shi, S.; Wang, Q.; Xu, P.; Chu, X. Benchmarking state-of-the-art deep learning software tools. In Proceedings of the 2016 7th International Conference on Cloud Computing and Big Data (CCBD), Macau, China, 16–18 November 2016; pp. 99–104.
41. Van Moffaert, K.; Drugan, M.M.; Nowé, A. Scalarized multi-objective reinforcement learning: Novel design techniques. In Proceedings of the 2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), Singapore, 16–19 April 2013; pp. 191–199.
42. 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; pp. 26–35.
43. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.

44. Watkins, C.J.; Dayan, P. Q-learning. *Mach. Learn.* **1992**, *8*, 279–292. [CrossRef]

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).