An Energy-Efficient Hardware Accelerator for Differentiable Neural Architecture Search on Edge Platforms

Mustafa Canıtez[1], Ahmet Emre Eser[1], Emirhan Özdemir[1]
Supervisor: Prof. Dr. Özcan Öztürk[1]

[1] Faculty of Engineering and Natural Sciences, Sabanci University

Abstract

Neural Architecture Search (NAS) automates the design of high-performing deep neural networks, however its high computational cost makes it impractical for resource-constrained edge devices. One of the most promising attempts to address this issue is Differentiable Architecture Search (DARTS). Existing accelerators such as GPUs are not optimized for DARTS' unique, bi-level optimization workload. This paper presents the design of a hardware accelerator targeting DARTS specifically. Our work establishes a architectural blueprint and demonstrates the feasibility of performing DARTS directly on edge platforms, enabling energy-efficient, highly performant intelligent systems with less expertise dependency.

1 Introduction

Choosing a suitable neural network architecture to train and deploy is an important concern in machine learning (ML), as the choice greatly affects the success of the resulting network after training. A suitable architecture must not only maintain a high accuracy and low latency, but it should also keep the number of parameters small to be compatible with resource-constrained edge environments. However, modern deep networks are chosen from an exponentially large search space. As ML applications diversify and architectures increase in complexity, manually designing a performant and scalable architecture has become exceptionally difficult. The architecture choice conventionally relies on expertise and it is error-prone. To make this process easier and more reliable, automation solutions have been proposed.

The field of Automated Machine Learning (AutoML) aims to automate the design of high-performing models. Its history dates back to early work on hyperparameter optimization (HPO) in the 1990s [1]. Neural Architecture Search (NAS), a relatively newer branch of AutoML solutions, automates the discovery of high-performing neural-network architectures from a pre-defined search space, but obtaining state-of-the-art architectures demands enormous computing power—up to 800 high-end GPUs for two weeks in early work by Zoph and Le [2].

Since Neural Architecture Search became mainstream after the original paper from 2016 [2], various improvements have been proposed. Even though those algorithms are branches of NAS and share the same goal, it is very crucial to note that their approaches to the problem substantially differ from each other. One of the most prominent methods proposed is Differentiable Architecture Search (DARTS), which decreases the search time to a few GPU-days through gradient-based optimization [3]. However, current ML accelerators—GPUs, TPUs [4], and generic AI cores—are optimized for inference or conventional training workloads and do not accommodate for forward passes, backward passes, and architecture-parameter updates of multiple cells that characterize DARTS. Consequently, existing hardware fails to meet the energy efficiency and time requirements of training in edge environments.

Our goal is to bridge this gap by designing, implementing, and evaluating a purpose-built hardware accelerator that executes the DARTS search loop within the strict power and latency budgets of edge platforms. By co-optimizing algorithm, micro-architecture, and tool-chain integration we aim to substantially reduce energy demands of NAS relative to a contemporary GPU baseline, while adhering to the strict latency constraints of on-device applications. We can thereby enable the use of NAS for resource-limited applications and help reduce the reliance on high-end GPUs.

2 Methodology

The DARTS framework can generate architectures for both recurrent (RNN) and convolutional (CNN) neural networks. However, the underlying operations and dataflow patterns for these two network types are substantially different. A unified hardware accelerator designed to support both would inevitably

incur significant overheads in terms of silicon area, power consumption, and design complexity. To create a highly optimized and efficient accelerator, we therefore chose to specialize in one domain. We selected the CNN variant for two primary reasons. First, the highly parallel and regular structure of convolutional computations is well-suited for FPGA and ASIC implementations [5]. Second, CNNs remain the cornerstone for a vast array of prominent edge-computing tasks, including image classification and object detection [5], making an optimized CNN search accelerator broadly applicable.

Our methodology is a hardware-software co-design process that maps the convolutional variant of DARTS [3] onto a purpose-built accelerator. The design is implemented in SystemVerilog and prototyped on a Xilinx UltraScale-class FPGA, chosen for its extensive resources which allow for implementation without significant precision loss. Our workflow consists of several sequential phases: algorithm profiling, micro-architecture definition, RTL implementation and simulation, and finally synthesis followed by software integration. While the FPGA is ideal for validation, we note that a future ASIC implementation could offer superior cost and power efficiency for deployment at scale.

2.1 Candidate Operation Set

To ensure our results are directly comparable to the baseline software implementation, we adopt the candidate operation set from the original DARTS paper for the CIFAR-10 search space [3]. Our accelerator is designed to execute the operation set O of seven primitive operations:

- 3×3 separable convolution
- 5×5 separable convolution
- 3×3 dilated separable convolution
- 5×5 dilated separable convolution
- 3×3 max-pooling
- 3×3 average-pooling
- Identity
- Zero

ReLU-Conv-BN order is preferred in DARTS for each convolution operation. All those operations are embedded in our ALU unit.

2.2 Numerical Precision

All tensors are represented in IEEE-754 half precision (FP16) [6]. This format was selected as an effective compromise, retaining numerical accuracy close to that of single-precision (FP32) while halving the memory footprint and doubling the potential density of multiply-accumulate (MAC) units on the target FPGA.

2.3 DARTS Training Loop

The core of the DARTS algorithm is to relax discrete architectural choices into a continuous search space. This is achieved by constructing an over-parameterized "super-network" where each edge between computational nodes represents a weighted mixture of all candidate operations (Section 2.1). The influence of each operation is controlled by a trainable architecture parameter, denoted as α. Proportional value of α simply represents the share of that operation among others.

The search process is an iterative bi-level optimization that alternates between updating these architecture parameters and the network's weights, denoted as ω. Following the procedure from the original DARTS paper [3], each iteration involves two gradient descent steps.

$$\nabla_\alpha \mathcal{L}_{val}(w^*(\alpha), \alpha) \approx \tag{1}$$

$$\nabla_\alpha \mathcal{L}_{val}\left(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha\right) \tag{2}$$

First, the architecture parameters α are updated by descending the gradient of the validation loss. This gradient calculation can be configured to use a first order approximation by adjusting the learning rate $\xi = 0$. This approach greatly reduces the required computing power, with the trade-off being slightly lower accuracy. The choice of ξ does not alter the high-level design, but it affects the design of the control unit. Second, the network weights ω (convolution filters for CNN training) are updated by descending the gradient of the training loss $\nabla_w \mathcal{L}_{train}(w, \alpha)$. This cycle of updating architecture and then weights is repeated until convergence. Our hardware accelerator is purpose-built to execute this sequence of forward and backward passes on both validation and training data, enabling the entire search process to be performed efficiently on an edge device.

3 Architecture

3.1 High Level Architecture

The high-level architecture of the accelerator is presented in Figure 1:

Nodes The intermediate feature maps of the DARTS super-network, corresponding to the computational nodes, are stored in a dedicated on-chip register file. This choice was made to provide high-bandwidth, low-latency access, which is critical for the frequent

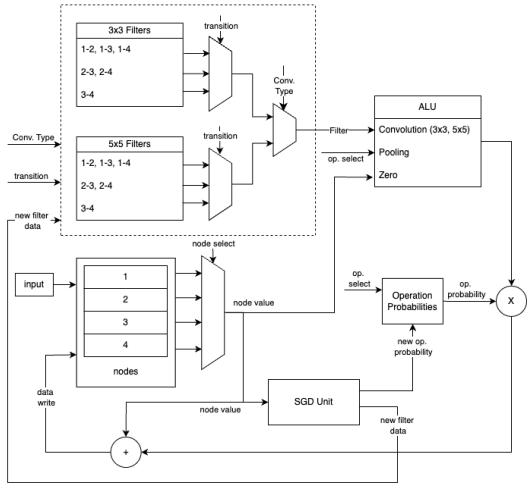

Figure 1: *High-level view of the accelerator architecture*

read-and-accumulate operations that characterize the DARTS forward pass.

ALU The ALU is the computational core of the accelerator, designed to execute the seven primitive operations defined in Section 2.1. For the four convolution operations, we implement a spatial architecture leveraging a systolic array of Processing Elements (PEs) to maximize throughput and data reuse. This design is highly efficient for the regular dataflow of convolutions. The pooling, identity, and zero operations are to be handled by simpler, dedicated data paths within the ALU.

SGD Unit The back-propagation stage utilizes a hardware stochastic gradient descent unit. Our unit is decided to calculate the gradient descent with 3, as demonstrated in [7].

$$\gamma \sum_{i=1}^{m} (\hat{y} - y) \otimes S_2 \tag{3}$$

In equation 3 S_2 is the input to the layer, $\otimes$ is the outer product operation, m is the number of samples in a training mini-batch and γ is the learning rate. The applicability of the formula above to SGD unit has not yet been studied.

Memory Subsystem The main memory will be used to store:

1. Filter weights
2. Operation probabilities/weights

Memory accesses during training runs are highly predictable. As a result, the system does not require prefetchers or complex cache hierarchies. Instead, an FSM will be used to queue subsequent accesses and ensure that they're fetched from memory into the registers or LUTs before they are requested from the computational unit.

3.2 Processing Element Array Micro-Architecture

A unified processing element (PE) array forms the computational core of our accelerator, designed to efficiently execute the matrix multiplication operations that dominate the convolutional layers. We adopt a systolic array architecture, a design proven to exploit the high degree of parallelism inherent in these workloads [8].

We chose to construct the PE array of the accelerator around a systolic array formation. Systolic array-based matrix multiplication units feature many processing elements in a grid structure and exploit the parallelism inherent in matrix multiplications by computing independent sub-operations simultaneously[8]. Parallel processing of intermediate products allows for time-efficient computation, which is one of our key improvement goals.

The data-flow pattern dictates the order of data movements across PEs by specifying which data types will be stored in the scratch-pads of PEs[8]. An overview of common data-flow strategies is provided below:

- **Weight Stationary (WS)** — keeps weights stationary while moving input activations and partial sums.
- **Output Stationary (OS)** — keeps partial sums stationary while moving weights and input activations between PEs.
- **Input Stationary (IS)** — keeps input activations stationary while moving weights and partial sums between PEs.
- **Row Stationary (RS)** — keeps rows of weights, input activations and partial sums stationary.

We have decided to move forward with RS since DARTS features numerous convolutional layers. The efficiency of RS for convolution operations has been demonstrated previously by CNN accelerators such as Eyeriss[9].

Our PE micro-architecture, shown in Figure 3, is inspired by the Eyeriss design but is explicitly adapted for an FPGA platform. Whereas an ASIC like Eyeriss can utilize custom, high-speed SRAM scratchpads within each PE, our FPGA-based design allocates registers and LUTs for the same purpose.

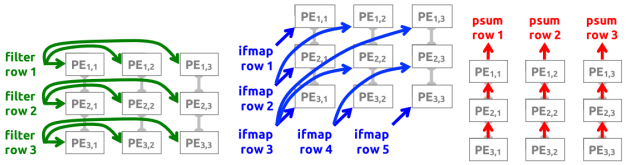

Figure 2: *Row-stationary data-flow in Eyeriss[9].*

The dataflow of Eyeriss is depicted in the Figure above. We are utilizing the same flow for the convolutional cells.

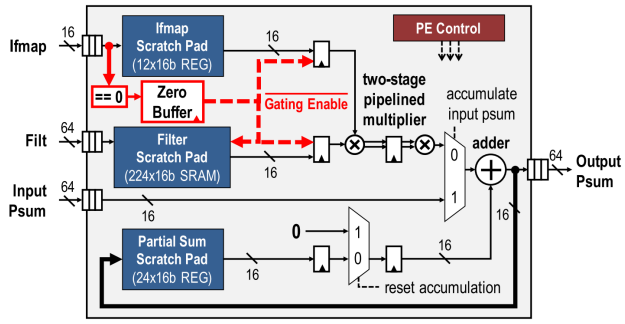

Figure 3: *Processing-element architecture of Eyeriss[9]. Our design follows a similar structure adapted for FPGA resources.*

Figure 3 demonstrates the general architecture of PEs used in Eyeriss. Since we are also implementing a very similar dataflow, our processing elements for convolution are also similar to Eyeriss. However, Eyeriss accelerator is an ASIC implementation and they prefer to use fast and small SRAM's as scratch pads. Since our FPGAs naturally do not include such architecture, we utilize registers and look-up tables for the same purpose.

PE Memory The acceleration of neural networks does not require complex memory hierarchies due to the predictable nature of memory accesses. Instead, accelerators in general rely on scratch-pad memory structures. Scratch-pad memory (SPM) is explicitly controlled by the software running on the accelerator. Unlike caches, where memory accesses automatically trigger a cycle of reads from the main memory followed by the prefetching of correlated addresses, SPMs have to be explicitly instructed to fetch and discard data. Each PE in our accelerator is planned to contain a small SPM where weights, partial sums and input activations are stored.

3.3 Control Architecture

The accelerator by itself is not Turing-complete in the sense that it lacks any branching/conditional instructions. Instead, instructions will be explicitly streamed by the master processor to the accelerator via a FIFO structure. Hence, the accelerator is essentially a specialized and powerful extension of the host system's CPU. The master–slave relationship between the accelerator and the system's CPU has led us to prioritizing FPGA platforms with processors embedded into the FPGA fabric.

3.4 Exploiting Sparsity

Sparsity (ie. having many repeated zeros in the data) arises due to the use of activation functions such as ReLU or Sigmoid and/or the use of pruning (setting small weights to zero after training) [8]. Additionally backpropagation might create sparse gradients if the activations are sparse as well [8]. DARTS makes extensive use of the ReLU for CNNs [3], hence the accelerator benefits greatly from exploiting sparsity.

Strategies such as zero skipping and compressed memory accesses [10] helps the accelerator achieve better performance and energy efficiency. Zero skipping is a strategy that involves skipping multiplication operations altogether when at least one operand is zero [10]. Compressed memory access strategy makes use of formats such as "compressed sparse row" or "compressed sparse column" to reduce the total size of sparse matrices in memory [10]. Consequently, less data have to be read from memory, resulting in better performance and energy efficiency. Zero skipping requires an extension to the finite state machine inside the systolic array controller, and the compressed format necessitates an intermediary unpacking unit between the memory management unit and the compute cores. Given how DARTS' format makes sparsity very likely to arise, we consider these extensions to be worthwhile.

4 Implementation Progress

Presented below is the implementation progress of the planned accelerator architecture which is composed of the following module hierarchy:

Table 1: *Implementation Status of Core Accelerator Modules*

Module	Status
Computation	
IEEE754 FP16 Adder	Implemented and tested
IEEE754 FP16 Multiplier	Tested
Singular Processing Element	Implemented
PE Array for Convolution	Incomplete
Pooling Unit	Not Implemented
Batch Normalization	Not Implemented
SGD Unit	Not Implemented
Control	
Control Unit (Main FSM)	Not Implemented
Memory Subsystem	
BRAM Module	Not Implemented
Memory Controller	Not Implemented

ALU Individual PE implementation is complete but the large PE array, that is to compute 3×3 and 5×5

convolutions is currently not finished. Pooling and zero skipping are trivial to implement but they were left to latter stages of development. A batch normalization unit will be placed to the output of the convolution units. BN calculation will be coupled from the systolic array as BN relies on statistics of the entire batch which could be more efficiently obtained from outside the systolic array. ReLU calculation, on the other hand, will be done inside PEs as it incurs minimal hardware requirements and will be applied on individual filter weights before convolution calculation.

SGD Unit A base gradient descent (non-stochastic) has been evaluated. A hardware algorithm specifically targeted at SGD is not known at the moment. Further evaluation of the adaptability of the discovered GD algorithm to SGD and DARTS is necessary.

Memory Subsystem Memory subsystem refers to system memory and should not be confused with local scratchpad memory of the processing elements. In its current stage, only the distribution of data between registers and main memory has been planned. Implementation for the memory subsystem has not yet begun. As discussed above, node accumulate values will be kept in on-chip hardware registers as they will be frequently updated and do not occupy large amounts of memory. The system requires a memory controller to ensure that values required for the rounds to follow are present before they are requested. The highly predictable nature of data reads during training rounds eliminates the need for cache hierarchies and prefetchers. This FSM at the current stage is not implemented.

Command Streaming Interface Theoretically the accelerator could work with any processor provided that the streamed instructions and their timing are valid. The embedded processor of an FPGA system such as AMD Xilinx Zynq has been determined to be the best candidate among the processors surveyed.

5 Implementation Logistics

5.1 RTL Development and Simulation

RTL development was done in the Verilog hardware description language. We used Verilator extensively to simulate our design. Verilator was selected for the initial phases of the project since the complexity of the simulated hardware was expected to be rather manageable. An inquiry into the performance issues of common hardware simulation tools revealed that HDL-to-software conversion tools, such as Verilator, do not perform any logic optimizations, which creates performance issues for large designs. Hence, the project will prioritize using logic simulators such as Vivado or Synopsys DC in its later stages.

```
Time: 8


Inputs:
        a[sign]: 0 a[exp]: 11110 a[mnts]: 0000000000
        b[sign]: 0 b[exp]: 11011 b[mnts]: 1111111111
        ADD/SUB: 0
Outputs:
        dut->o_y = 0111100100000000
Split:
        out[sign]: 0x0, out[exp]: 0x1e, out[mnts]: 0x100
        out[sign]: 0, out[exp]: 11110, out[mnts]: 0100000000
============================================================
Time: 9


Inputs:
        a[sign]: 0 a[exp]: 11110 a[mnts]: 0000000000
        b[sign]: 0 b[exp]: 11011 b[mnts]: 1111111110
        ADD/SUB: 0
Outputs:
        dut->o_y = 0111100100000000
Split:
        out[sign]: 0x0, out[exp]: 0x1e, out[mnts]: 0x100
        out[sign]: 0, out[exp]: 11110, out[mnts]: 0100000000
============================================================
```

Figure 4: *Sample Verilator simulation output for the adder module.*

5.2 Deployment and Evaluation

Once the accelerator is complete, our goal is to deploy it on our UltraScale Xilinx FPGA and run DARTS until convergence for various datasets. This way, we can select the best candidates and train them from scratch to see how well they actually perform. Finally, we are planning to compare the output neural network's results with state-of-the-art (SOTA) architectures; the output of DARTS is a neural network, and the results being compared with the SOTA will be the performance of the output network. This way, we can ensure the results are actually competitive with the best current options.

Another important evaluation criterion is the power consumption of the design. However, power measurements obtained on FPGAs are not reliable. Until an ASIC tape-out is available, simulating the transistor-level schematic of the design seems to be the most accurate option for power metrics.

6 Conclusion and Future Work

This project demonstrates the technical feasibility and potential impact of a hardware accelerator for Differentiable Architecture Search. However, the implementation is not yet complete and requires much more work before we can actually compare our results to a regular high-end GPU setup.

To the best of our knowledge this project constitutes the first attempt at the construction of a specialized neural architecture search accelerator as of 2025.

Hence, it is not possible to determine the SOTA in the domain of NAS acceleration on hardware. Additionally, most accelerator architectures are influenced by the demands of their target algorithms to a great extent, which makes them quite distinct in terms of their capabilities.

Several limitations in addition to the implementation remain at this stage. The current prototype still relies on embedded ARM cores for instruction streaming. Power measurements on FPGA cannot accurately predict ASIC efficiency, and the design has yet to pass comprehensive corner-case verification for denormalized floating-point numbers and overflow handling.

The implementation phase, once completed, is planned to be followed by an extension targeting RNNs, which are also within DARTS' domain. However, as discussed in the previous sections, performance gains in RNN training will be sub-par compared to that of CNNs. An ASIC implementation is necessary to obtain accurate power metrics, which in turn are necessary to determine the overall success of the accelerator.

References

[1] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, eds. *Automated Machine Learning. Methods, Systems, Challenges*. 1st ed. Computer Science, Computer Science (R0). 9 b/w illustrations, 45 illustrations in colour. Published eBook: 17 May 2019. Hardcover: 28 May 2019. Springer Cham, 2019. XIV+219. ISBN: 978-3-030-05317-8. DOI: 10.1007/978-3-030-05318-5. URL: https://doi.org/10.1007/978-3-030-05318-5.

[2] Barret Zoph and Quoc V. Le. *Neural Architecture Search with Reinforcement Learning*. 2017. arXiv: 1611.01578 [cs.LG]. URL: https://arxiv.org/abs/1611.01578.

[3] Hanxiao Liu, Karen Simonyan, and Yiming Yang. *DARTS: Differentiable Architecture Search*. 2019. arXiv: 1806.09055 [cs.LG]. URL: https://arxiv.org/abs/1806.09055.

[4] Norman P. Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 1–12. DOI: 10.1145/3079856.3080246.

[5] Feng Yan, Andreas Koch, and Oliver Sinnen. *A survey on FPGA-based accelerator for ML models*. 2024. arXiv: 2412.15666 [cs.AR]. URL: https://arxiv.org/abs/2412.15666.

[6] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[7] Yufeng Hao. *A General Neural Network Hardware Architecture on FPGA*. 2017. arXiv: 1711.05860 [cs.CV]. URL: https://arxiv.org/abs/1711.05860.

[8] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks*. Springer, 2020.

[9] Yu-Hsin Chen et al. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357.

[10] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. 2017. arXiv: 1703.09039 [cs.CV]. URL: https://arxiv.org/abs/1703.09039.