SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures

Alexander Montgomerie-Corcoran*, Zhewen Yu* and Christos-Savvas Bouganis
Dept. of Electrical & Electronic Engineering
Imperial College London, UK
{alexander.montgomerie-corcoran15, zhewen.yu18, christos-savvas.bouganis}@imperial.ac.uk

***Abstract*—Toolflows that map Convolutional Neural Network (CNN) models to Field Programmable Gate Arrays (FPGAs) have been an important tool in accelerating a range of applications across different deployment settings. However, the significance of the problem of finding an optimal mapping is often overlooked, with the expectation that the end user will tune their generated hardware to their desired platform. This is particularly prominent within Streaming Architectures toolflows, where there is a large design space to explore. There have been many Streaming Architectures proposed [1]–[3], however apart from fpgaConvNet [1], there is limited support for optimisation methods that explore both performance objectives and platform constraints. In this work, we establish a framework, SAMO: a Streaming Architecture Mapping Optimiser, which generalises the optimisation problem of mapping Streaming Architectures to FPGA platforms. We also implement both Brute Force and Simulated Annealing optimisation methods in order to generate valid, high performance designs for a range of target platforms and CNN models. We are able to observe a 4x increase in performance compared to example designs for the popular Streaming Architecture framework FINN [3].**

I. Introduction

The success of deep learning models represented by CNNs has greatly motivated the research into hardware accelerators designed for these models. A popular class of platforms for these accelerators are FPGAs, as their versatility and range of sizes suit a variety of applications. When deployed on these re-configurable platforms, accelerators can be tailored for different CNN workloads in order to achieve high performance. Therefore, many toolflows have been proposed in order to map CNNs to FPGAs, and they mainly focus on two types of architectures: Systolic Array and Streaming (also known as dataflow) [4].

A Systolic Array architecture design typically contains an array of processing elements which are designed to accelerate a matrix multiplication operation. The CNN model is executed by mapping convolution layers to matrix multiplication operations, leading to the processing elements of the architecture being time-shared across the layers. Systolic Array architectures are therefore very flexible, as they have the ability to map nearly any CNN model to a single hardware design. Because of this flexibility, these architectures are often served as compute kernels in general-purpose neural network accelerators that are not tailored to specific CNN models [5], [6].

Streaming Architectures, on the contrary, tailor the generated hardware design towards the computation and memory workload of a specific CNN model. Instead of time-sharing processing elements between layers, each layer has custom hardware specific to it, which is all fitted onto a given FPGA platform simultaneously. The computation kernels for each layer are then pipelined. This tailoring of the hardware to a platform means that Streaming Architectures have high throughput and energy efficiency [7] compared to equivalent Systolic Array architectures. However, the design process for a Streaming Architecture-based accelerator is often more time consuming and tedious, as the customisability brings a large design space to explore. Unlike Systolic Array architectures where the design space exploration can be carried out by brute force enumeration with the help of an analytical resource model [8], or a roofline model [9], the Streaming Architecture requires a more informed method for efficiently exploring the design space.

Early work on Streaming Architecture accelerators explored the design space manually, resulting in sub-optimal designs that did not make efficient use of their platform's resources [2], [10]. Several works have proposed algorithms that allow automatic exploration of the design space [11], [12], by following certain guidelines and rules that contribute to a efficient design. However, these guidelines and rules are closely coupled to the proposed accelerator's building blocks, which restricts their ability to generalise the CNN to architecture mapping problem for other architectures. This in turn restricts the ability to explore improved optimisation methods, as the optimisation problem is closely tied to the specific accelerator framework.

In this work, we present SAMO[1]: a Streaming Architecture Mapping Optimiser. We generalise the CNN to Streaming Architecture mapping problem by proposing an abstract representation of the accelerator's building blocks, which is referred to as the Hardware Description graph. With this abstraction in place, we are able to define variables, constraints and objectives for the optimisation problem, which leads to a framework for exploring different optimisation methods. Both a Brute Force and Simulated Annealing optimiser are implemented in this work in order to demonstrate the potential for exploring this design space. The framework is integrated into popular Streaming Architectures (fpgaConvNet, FINN and HLS4ML), and used to increase the performance of base designs for a range of CNN models and FPGA platforms.

*equal contribution

[1]https://github.com/AlexMontgomerie/samo

II. Background

In this work, we consider three toolflows: fpgaConvNet [1], FINN [3] and HLS4ML [2], which are able to map a given CNN model to a target FPGA device, by generating a Streaming Architecture-based accelerator which optimises latency. These three toolflows were chosen due to their popularity as Streaming Architectures, as we as their variations in design space, which demonstrates the generalisation abilities of the SAMO tool. A brief overview of these toolflows are given in Table I.

	fpgaConvNet	**FINN**	**HLS4ML**
Design Space	Large	Medium	Small
Existing Optimiser	Simulated Annealing	Rule-based	N/A

TABLE I: Comparison on Streaming Architecture toolflows.

The FINN toolflow is able to generate Streaming Architecture accelerators for low-precision Quantised Neural Networks (QNNs) [7]. FINN was originally customised for Binarised Neural Networks (BNNs) but has now been extended to support other fixed-point data types [13]. Most fixed-point operations, including quantised convolutional layers and quantised fully-connected layers are implemented by the Matrix-Vector Threshold Unit (MVTU). Each MVTU contains multiple Processing Elements (PE) and SIMD lanes for parallel Matrix-Vector operations, followed by quantisation thresholding. The values for PE and SIMD inside each MVTU can be determined by hand-tuning or adopting a rule-based optimisation algorithm which allocates more resources to the slowest MVTU in the whole accelerator [3]. However, the authors of FINN also state this algorithm to be sub-optimal and can often be outperformed by hand-tuning[2].

HLS4ML [2] was originally developed for accelerating simple machine learning models in particle physics experiment, however it has been further developed for more general CNN model acceleration. HLS4ML sacrifices the configurability of the accelerator to obtain simple and low-latency accelerator designs [14]. Therefore, HLS4ML has a relatively small design space compared with other Streaming Architecture toolflows. Specifically, HLS4ML supports two hardware generation methods: *resource* and *latency*. For the *latency* method, the hardware is completely unrolled, and the HLS compiler is given the task of optimising the latency for a given initiation interval goal. And for the *resource* method, the hardware is only partially unrolled, and the *reuse-factor* parameter defines how often resources are re-used. This dictates the achievable parallelism for the *resource* designs.

The fpgaConvNet [1] toolflow is a streaming-based accelerator which focuses on configurability in order to achieve high performance for a range of networks and platforms. This toolflow exposes many degrees of parallelism in the network including *coarse-grained folding* and *fine-grained folding*. The toolflow further supports FPGA configuration through the partitioning and partial reloading of data in order to overcome resource constraints. Therefore, fpgaConvNet has the largest design space amongst the three considered toolflows. fpgaConvNet also has explored optimisation as part of the design flow, proposing a Simulated Annealing optimiser [15] to explore it's design space, however this has been highly tailored to this specific toolflow.

[2]https://github.com/Xilinx/finn/blob/main/src/finn/transformation/fpgadataflow/set_folding.py

III. Optimisation Problem

In this section, the CNN to Streaming Architecture mapping space will be defined, and the optimisation problem that surrounds this will be outlined. This abstraction of the mapping procedure and optimisation problem is the core backbone of the SAMO framework, and this section serves to express this representation.

The optimisation problem we wish to solve is of tailoring a Streaming Architecture to a specific CNN model and FPGA platform pair, with the desire to efficiently utilise the available resources on the platform in order to optimise the throughput for the given network. This requires knowledge of how parameters of the given accelerator affect both performance and resources. Furthermore, the optimisation variables must generalise for all Streaming Architectures, and so the design parameters for the architectures must be understood. Before defining the optimisation problem, we must first outline the assumptions for the design space. These assumptions are,

- The Streaming Architecture is pipelined
- Layers in the CNN models are sequential
- There exist models of latency and resource utilisation
- The Streaming Architecture targets a single platform

A. Hardware Description Graph

The mapping of layers of a CNN model to hardware building blocks of a Streaming Architecture can be described in terms of directed graphs. The CNN model can be described as a graph with L layers as $M = \{l_1, \dots l_L\}$, where l_i is a layer within the CNN model. This graph has edges E_M between the layers. As the focus of this work is on sequential networks, the edges are only between adjacent nodes, so $E_M = \{(l_1, l_2), \dots (l_{L-1}, l_L)\}$. Streaming architectures frameworks perform a mapping from a CNN model graph to a Hardware Description graph, where each layer in the CNN model gets mapped to a computation node or set of computation nodes in the Hardware Description graph. We can describe this Hardware Description graph as H, which contains N computation nodes, $H = \{n_1, \dots n_N\}$ and where n_i is a computation node within it. As the CNN models are sequential, so is the Hardware Description graph, and the edges of H are such that $E_H = \{(n_1, n_2), \dots (n_{N-1}, n_N)\}$.

B. Variables

This Hardware Description graph, H, is what we will performing our optimisations on and in order to do so, optimisation variables must be defined. For each node n_i, there are three associated variables: *input channel folding* (s_i^I), *output channel folding* (s_i^O) and *kernel folding* (k_i). These three variables are chosen to summarise the current landscape

of Streaming Architecture frameworks and their respective performance parameters.

The input and output channel folding variables describe the degree of parallelism of the channel dimension of the feature-map entering and exiting a node respectively. It is worth noting that although this could be extended to all the other dimensions of the feature-map, the channel dimensions have the fewest dependencies, and their parallelism is a lot more feasible to exploit. The kernel folding variable describes the parallelism of computations within a node. This typically relates to the kernel dot product operation found within convolution layers, and the degree of parallelism across the *kernel size*^{2} multiply operations. However, this can be expanded to generalise parallelism within any computation node, given the relevant constraints are in place.

C. Constraints

There exist both constraints on the Hardware Description graph, as well as the individual nodes within. In this subsection we define the constraints that are observed across all Streaming Architectures, although specific frameworks are not necessarily constrained by all.

One important constraint that exists across all current Streaming Architecture frameworks is that the channel folding variables are factors of the channel dimension of the feature-map that the node is operating on. This is described in Eq. (1), where c_i^I and c_i^O are the input and output channel dimensions of the feature-map of the i^{th} respectively.

$$\begin{aligned} c_i^I \bmod s_i^I = 0 \; \forall \; i \in \{1 \dots N\} \\ c_i^O \bmod s_i^O = 0 \; \forall \; i \in \{1 \dots N\} \end{aligned} \tag{1}$$

There will also exist a subset of nodes, such as the hardware for Max Pooling or ReLU layers, where the channel folding must match, as the output channel dimension depends on the input. This subset of nodes, $N' \subset N$, have this equality constraint described in Eq. (2), which will be referred to later as *intra folding matching*.

$$s_i^I = s_i^O \; \forall \; i \in \{1 \dots N'\} \tag{2}$$

For the Hardware Description graph itself, another common constraint that exists is the need to have matching folding factors between nodes in order to ensure that all the data lines are connected. This constraint is described in Eq. (3), and is referred to as *inter folding matching*.

$$s_i^O = s_{i+1}^I \; \forall \; i \in \{1 \dots N-1\} \tag{3}$$

D. Models

Before introducing the optimisation problem, let's describe the models which are required from the Streaming Architecture frameworks. These required models for each node n_i are the resource model, $r(n_i, s_i^I, s_i^O, k_i)$, and latency model, $t(n_i, s_i^I, s_i^O, k_i)$, which are a function of the optimisation variables. The resource model gives the utilisation of the various resource types on the FPGA platform. The latency model gives an estimation of the number of clock cycles needed to execute the hardware node.

These models can be used to construct the constraints and objective for the optimisation problem. In particular, we can use these node-level models to evaluate the resource utilisation ($R(H, E_H)$) and latency ($T(H, E_H)$) of the whole Hardware Description graph, as described in Eq. (4).

$$\begin{aligned} R(H, E_H) &= \sum_{i \in N} r(n_i, s_i^I, s_i^O, k_i) \\ T(H, E_H) &= \max\{t(n_i, s_i^I, s_i^O, k_i) : i = 1 \dots N\} \end{aligned} \tag{4}$$

As there is the assumption that the hardware is pipelined and the CNN models are sequential, the latency of the network is dictated by the slowest node in the graph[3]. The total resource utilisation is the sum of the resource utilisation of all nodes in the graph.

E. Objective

Now that the properties of the Hardware Description graph are established, the objective of the optimisation problem is outlined next. As there is an assumption that the whole CNN model must fit on the target FPGA platform, this means that latency is the most meaningful metric in terms of performance. We can describe the whole optimisation problem as is in Eq. (5), where $R_{platform}$ describes the resource constraints of the target platform.

$$\begin{aligned} \min_{s^I, s^O, k} \quad & T(H, E_H) \\ \text{s.t.} \quad & R(H, E_H) \leq R_{platform} \\ & C(H, E_H) = 0 \end{aligned} \tag{5}$$

With the whole optimisation problem outlined, the next step is applying this abstraction to an actual Streaming Architecture framework, and subsequently solving the optimisation problem through existing optimisation methods.

IV. Framework

The Hardware Description graph provides a unified representation of the CNN model and the accelerator design space. In this section, we demonstrate how this unified representation can be integrated into existing Streaming Architecture toolflows such as fpgaConvNet, FINN and HLS4ML, which are referred to as backends, in order to generate high performance accelerator designs. An overview of the framework is given in Figure 1.

A. Parser

The parser is responsible for converting the CNN model into the Hardware Description graph. This conversion starts with transforming the CNN model into a custom intermediate representation (customised IR). This customised IR is designed by the authors of the Streaming Architecture toolflow in order to map the CNN model to a hardware implementation. During

[3]Pipeline depth is ignored as it has a negligible affect on latency.

this step, the layers of the CNN model are replaced by tunable hardware building blocks from the accelerator framework, as Listings 1 to 2 show.

After the CNN model is mapped to the backend's customised IR, it is further unified into the Hardware Description graph through SAMO's provided wrappers. These wrappers abstract the customised IR both at the node and network levels. The node wrapper stores the tunable design parameters as well as the API of the resource and latency models belonging to each hardware building block (described in Listing 3). Therefore, the actual operation and implementation of the building block is hidden from the Hardware Description graph.

Default constraints are provided by SAMO, however the backends have their own assertions on the design parameters in order to validate their designs. These assertions can be translated to constraints for optimisation with simple integration into the Hardware Description graph.

Listing 1: CNN model

```
Network {
  nodes {
    l_1 = FCLayer_0
    l_2 = FCLayer_1
  }
}
```

Listing 2: Customised IR (FINN-ONNX)

```
Network {
  nodes {
    m_1 = Threshold_0
    m_2 = StreamFCLayer_0
    m_3 = StreamFCLayer_1
    m_4 = LabelSelect_0
  }
}
```

Listing 3: Hardware Description Graph

```
Network {
  nodes {
    n_1 = {m_1.rsc(), m_1.cycle(), m_1.PE}
    n_2 = {m_2.rsc(), m_2.cycle(), m_2.SIMD, m_2.PE}
    n_3 = {m_3.rsc(), m_3.cycle(), m_3.SIMD, m_3.PE}
    n_4 = {m_4.rsc(), m_4.cycle(), m_4.PE}
  }
}
```

B. Optimiser

After the Hardware Description graph is obtained for the given backend, next the optimisation problem needs to be solved. There are two optimiser algorithms that have been considered in this work,

- **Brute-force search**, which enumerates all possible combinations of design parameters. Any design point that violates the constraints of the Hardware Description graph will be ignored. This algorithm can find the optimal design point and minimise the accelerator latency at a cost of lengthy searching time.
- **Simulated annealing** [16], which is a stochastic searching algorithm. All the optimisation variables $V = \{s^I, s^O, k\}$ are initialised to their minimum at the starting point, which corresponds to the resource-minimal accelerator design with the greatest latency. During each iteration of the search, these optimisation variables are randomly updated to V^{new}. This update will be accepted only when the constraints of the Hardware Description graph are met and (6) is also satisfied. Otherwise, this update is discarded and the Hardware Description graph is reverted to it's state during the last iteration.

$$exp(min(0, \frac{T - T^{new}}{K})) \geq x \sim U(0,1) \qquad (6)$$

T and T^{new} correspond to the latency before and after this update respectively. K is a hyper-parameter, also known as temperature, that decays linearly over iterations. x is sampled from a uniform distribution $U(0,1)$. Equation (6) reflects the probability that an update is accepted. When the latency T^{new} is smaller than T, (6) is always satisfied. Otherwise, the optimiser accepts this update with a probability relating to the difference between T^{new} and T as well as the temperature K.

The Simulated Annealing optimiser implemented in SAMO will also attempt to fix any constraint violations during each step. The occurrence of constraint violations is due to the variables $V = \{s^I, s^O, k\}$ being optimised independently in each iteration, which ignores any dependencies between them. When an optimisation variable is updated, our optimiser will propagate these changes through the whole graph. Therefore, any conflict between optimisation variables that causes a constraint violation is resolved during this propagation.

C. Exporter

The optimised Hardware Description graph is transformed back to the customised IR of each backend, and is used to configure the hardware building blocks with the optimised design parameters. The configured hardware building blocks are converted into synthesisable hardware description code, and this is used to generate bitstreams that can be programmed on to the target FPGA platform.

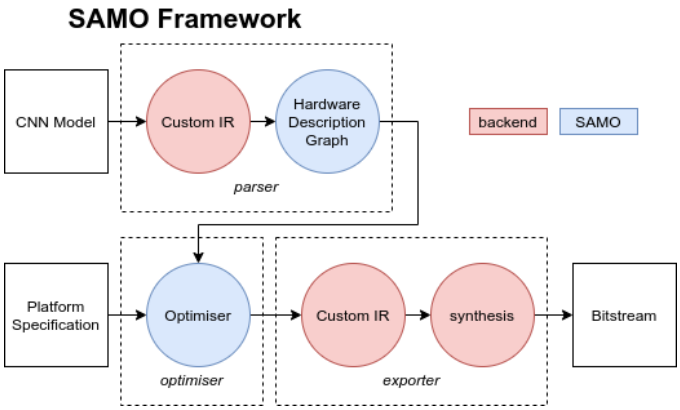


Fig. 1: Overview of the proposed SAMO framework

V. Evaluation

In this section, we will describe how the chosen backends have been integrated into the SAMO framework, and evaluate the framework's potential to find optimal designs using both Brute Force and Simulated Annealing optimisation methods.

We have evaluated SAMO on three different open-source Streaming Architectures: HLS4ML [2], FINN [3] and fpgaConvNet [15]. Although all these backends share the fact that they are Streaming Architectures, each of them have different properties when it comes to their design parameters and constraints. The relationship between the parameters for each backend and the respective optimiser variables of the SAMO framework are given in Table II.

Variable	Backend		
	fpgaConvNet	*FINN*	*HLS4ML*
Input Channel Folding	Coarse-In	SIMD	Reuse-Factor
Kernel Folding	Fine		
Output Channel Folding	Coarse-Out	PE	

TABLE II: Relationship between backend parameters and SAMO optimiser variables.

For HLS4ML, all possible degrees of parallelism are summarised within one tunable parameter, as in the backend the operations are fully unrolled and the HLS tool uses the *reuse-factor* as a design target for latency. FINN combines *input channel folding* and *kernel folding* together, as the kernel dot product dimensions are combined into the channel dimension of the incoming feature-map for their computation nodes. The fpgaConvNet toolflow has explicit design parameters for each of the given optimiser variables.

In terms of constraints, Table III summarises which constraints apply to the different backends. All of them follow the constraints on the channel folding variable being a factor of the channel dimension.

Constraint	fpgaConvNet	FINN	HLS4ML
intra channel matching	✓	✓	✗
inter channel matching	✗	✓	✓

TABLE III: Constraints on the backends.

In addition to these constraints, there are constraints defined on the kernel folding. For fpgaConvNet, this is a factor of the kernel size of the respective convolution layer. With HLS4ML, the kernel size captures the complete achievable parallelism of the unrolled node as all the optimisation variables are combined into a single design parameter. However, the HLS4ML tool ensures that the *inter channel matching* constraint is met.

In terms of latency and resource models, currently only fpgaConvNet and FINN[4] have both. For the HLS4ML tool, we have provided a latency model as well as a model for DSP utilisation based on observations of the relationship between their *reuse-factor* parameter and resource utilisation and latency.

For the evaluation, we have decided on four CNN models to evaluate the effectiveness of the framework. These models are described in Table IV, which gives an indication of the size and complexity of the CNN model, all of which are taken from examples given by the evaluated backends.

[4]FINN does not support Flip Flop resource estimation.

Network	Parameters	No. Conv	No. Dense
4-layer	4K	0	4
TFC	60K	0	4
LeNet	430K	2	2
CNV	1.5M	6	3

TABLE IV: CNN models for evaluation.

A. Brute Force

As described in Section IV, a Brute Force optimisation method has been designed that explores the entire design space in order to determine the design with the greatest performance. In Table V, a comparison of runs for different networks is shown. The table highlights the size of the unconstrained design space (referred to as the *problem size*), the number of designs evaluated per second, and the estimated time to evaluate the whole design space.

Network	Backend	Problem Size	Iter. /s	Est. Eval. Time
4-layer	HLS4ML	$1.02\text{x}10^{4}$	25000	407 ms
	FINN	$3.18\text{x}10^{7}$	120	3 days
	fpgaConvNet	$6.35\text{x}10^{5}$	1000	10 min
TFC	HLS4ML	$5.46\text{x}10^{8}$	15000	10 hr
	FINN	$6.35\text{x}10^{9}$	125	588 days
	fpgaConvNet	$1.33\text{x}10^{13}$	900	47 decades
LeNet	HLS4ML	$6.76\text{x}10^{8}$	14000	13 hr
	FINN	$5.02\text{x}10^{12}$	110	145 decades
	fpgaConvNet	$2.01\text{x}10^{13}$	600	106 decades
CNV	HLS4ML	$6.32\text{x}10^{24}$	6000	$3.3\text{x}10^{11}$ centuries
	FINN	$1.95\text{x}10^{35}$	50	$1.2\text{x}10^{24}$ centuries
	fpgaConvNet	$1.22\text{x}10^{38}$	300	$1.3\text{x}10^{26}$ centuries

TABLE V: Comparison of Brute Force optimisation for the networks (*4-layer, TFC, LeNet* and *CNV*) and backends (*HLS4ML, FINN* and *fpgaConvNet*).

The table shows that each backend has different characteristics, in terms of both problem size and iterations per second. The reason for differences in problem size is due to the number of design parameters and constraints for the Streaming Architectures. For example, fpgaConvNet typically has a larger problem size due to not having the *inter channel matching* constraint as well as supporting all the optimisation variables. The iterations per second are due to both the Hardware Description graph's size as well as the complexity in evaluating resource and latency models. As the network size increases, more nodes must be evaluated at every iteration of the Brute Force optimiser.

Apart from the *4-layer* network, exploring the design space is intractable, with some designs being estimated to take centuries to evaluate. This is due to the large design space which scales with the size of the CNN model, as well as the limitations on the rate of evaluating these design points due to the complex latency and resource model evaluation as well as constraint evaluation. These results serve as the motivation for exploring more feasible optimisation methods.

Network	Platform	Precision	Latency (cycles)		Increase	Resource Utilisation (%)							
						DSP		BRAM		LUT		FF	
			Init.	*Opt.*		*Init.*	*Opt.*	*Init.*	*Opt.*	*Init.*	*Opt.*	*Init.*	*Opt.*
4-layer	*ZedBoard*	*w16a16*	2058	42	49x	1.8	89.5	2.5	6.4	20.6	23.6	16.4	17.4
TFC			1154785	1164	992x	1.8	95.5	23.9	18.2	76.1	132.5	75.1	116.4

(a) HLS4ML

Network	Platform	Precision	Latency (cycles)		Increase	Resource Utilisation (%)							
						DSP		BRAM		LUT		FF	
			Init.	*Opt.*		*Init.*	*Opt.*	*Init.*	*Opt.*	*Init.*	*Opt.*	*Init.*	*Opt.*
4-layer	*ZedBoard*	*w16a16*	2203	143	15x	1.8	31.4	4.6	3.2	12.7	45.4	8.1	31.1
TFC			50980	814	63x	1.8	40.5	13.6	33.2	21.3	164.1	12.8	94.5
LeNet	*ZC706*		1600066	8046	199x	0.6	33.6	9.7	41.2	12.8	67.7	8.0	44.9
CNV	*U250*		28901422	200749	144x	-	21.4	-	47.3	-	118.4	-	80.6

(b) fpgaConvNet

Network	Platform	Precision	Latency (cycles)		Increase	Resource Utilisation (%)							
						DSP		BRAM		LUT		FF	
			Init.	*Opt.*		*Init.*	*Opt.*	*Init.*	*Opt.*	*Init.*	*Opt.*	*Init.*	*Opt.*
4-layer †	*ZedBoard*	*w4a4*	2059	42	49x	1.8	75.9	2.1	3.9	14.9	29.1	9.0	13.1
TFC †			50176	403	125x	1.8	100	5.7	11.1	5.8	22.4	-	15.0
LeNet †	*ZC706*		1600000	14546	110x	0.4	36.0	13.2	15.3	12.1	11.8	-	6.1
CNV ‡	*U250*		28901376	28800	1004x	0.3	92.5	31.6	25.6	4.8	71.4	-	-
		w1a1	28901376	8196	3526x	0.0	0.0	0.0	0.0	9.6	82.1	-	-

(c) FINN

TABLE VI: Comparison of optimised designs to initial configurations using the Simulated Annealing optimiser for the networks (*4-layer, TFC, LeNet* and *CNV*) and backends (*HLS4ML, FINN* and *fpgaConvNet*).

† Values for initialised design based on the resource and latency models

‡ Values for initialised and optimised design based on the resource and latency models

B. Simulated Annealing

Now we turn to the Simulated Annealing optimiser, and evaluate the designs it is able to produce. In our experiment, the hyper-parameter temperature K is initialised as 1000 and it is reduced by 2% for every iteration. The optimisation will be terminated once K is reduced to 1.

In Figure 2, we can see the design space evaluated by the optimiser for the CNV model, and all the backends. The red cross indicates the final design which is chosen. The resource utilisation is described in Eq. (7). The predicted utilisation is for a U250 FPGA platform.

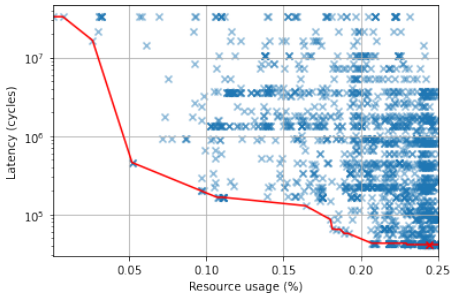
$$rsc = \frac{1}{4}\left(\frac{BRAM_{design}}{BRAM_{platform}} + \frac{DSP_{design}}{DSP_{platform}} + \frac{LUT_{design}}{LUT_{platform}} + \frac{FF_{design}}{FF_{platform}}\right) \tag{7}$$

From the figure, we can see the design space that has been explored. It is clear that the simulated annealing run is able to cut down searched design space significantly compared to brute force, with each run exploring around 6000 design points. Although it is difficult to evaluate whether these annealing runs achieve a global optimum, the pareto-optimal front suggests that the optimiser is able to search a large spectrum of meaningful designs and significantly reduce latency compared to the starting point.

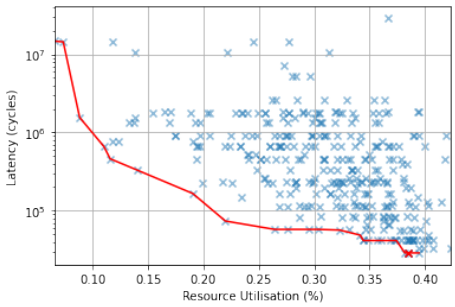
Having evaluated the design space that is explored by the Simulated Annealing optimiser, we can now evaluate the designs produced by the tool, which are shown in Table VI. In this table, we can see a comparison between a design from the initial starting point and the generated design point from Simulated Annealing, shown by the latency and resource utilisation from synthesis and implementation of these designs (unless stated otherwise). It is the author's understanding that convolution layers are not yet fully supported by HLS4ML, hence why the *LeNet* and *CNV* networks are not evaluated for this backend. For running the *CNV* network with FINN, we are aware that it is supported, however we were not able to generate designs due to the Vivado HLS tool hanging, despite validating the generated designs using the FINN toolflow. We were also not able to obtain bitstreams for initial configurations of FINN, due to issues with the HLS tool.

This table highlights the power of automated design space exploration in order to achieve high performance designs. For example, we are able to achieve a performance increase of 1000x compared to a baseline design for the HLS4ML toolflow. It's shown that using the SAMO tool, users will be able to achieve significant latency reduction with only being required to provide a CNN model and target platform. The table also shows the resource utilisation of the target platform for the generated designs. It can be seen that the SAMO tool aims for as much utilisation as possible, however may over utilise resources due to inaccuracies in the backend's resource models. For the HLS4ML backend, this is due to the fact that there is only a DSP model, so there is no understanding of LUT, BRAM or FF resources. With fpgaConvNet, the tool struggles with modelling BRAM utilisation as it over-estimates, and so observed BRAM utilisation is often a lot lower than predicted whilst LUT utilisation is a lot higher. Conversely, FINN benefits from quite an accurate resource model, and alongside the low precision it supports, it is able to

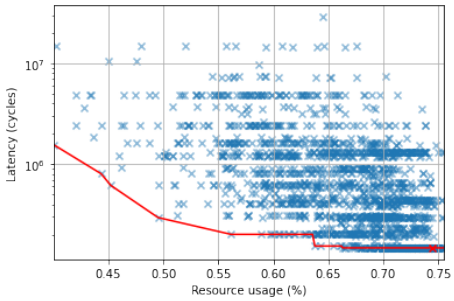
achieve high performance designs which stay within resource limits. A particularly interesting design is that of *CNV* for FINN with a *w1a1* precision. The example design that the FINN creators give for this CNN model achieves a latency of roughly 33000 cycles[5], while the SAMO tool is able to find a valid design that has a predicted latency of 8192 cycles, leading to a 4x increase in performance compared to the hand-tuned design.



(a) HLS4ML



(b) FINN



(c) fpgaConvNet

Fig. 2: Pareto-optimal front for the CNV network using the Simulated Annealing optimiser. The red line indicates the pareto-optimal front. The blue markers indicate the valid design points explored, and the red marker indicates the chosen design point.

VI. Conclusion

This paper presents the SAMO framework, a Streaming Architecture Mapping Optimiser, which serves as a starting point for research into optimisation methods for this class of CNN accelerators. We evaluated the potential of Simulated Annealing as an optimiser and have seen considerable gains in performance using this. The framework has been integrated with popular open-source Streaming Architectures in order to demonstrate it's ability to achieve high performance designs across a range of FPGA platforms. With this framework established we hope to further explore improved optimisation methods as well as different application settings, such as the Neural Architecture Search (NAS) co-design space.

[5]https://github.com/Xilinx/finn-examples/blob/main/finn_examples/notebooks/1_cifar10_with_cnv_networks.ipynb

References

[1] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, 2019.

[2] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, Jul. 2018.

[3] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, 2018.

[4] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *arXiv preprint arXiv:1803.05900*, 2018.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[6] D. A. Vink, A. Rajagopal, S. I. Venieris, and C.-S. Bouganis, "Caffe barista: Brewing caffe with fpgas in the training loop," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020.

[7] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[8] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017.

[9] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[10] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated fpgas: Fpga framework for convolutional neural networks," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016.

[11] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016.

[12] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018.

[13] J. Faraone, G. Gambardella, D. Boland, N. Fraser, M. Blott, and P. H. Leong, "Customizing low-precision deep neural networks for fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.

[14] T. Aarrestad, V. Loncar, N. Ghielmetti, M. Pierini, S. Summers, J. Ngadiuba, C. Petersson, H. Linander, Y. Iiyama, G. Di Guglielmo *et al.*, "Fast convolutional neural networks on fpgas with hls4ml," *arXiv preprint arXiv:2101.05108*, 2021.

[15] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[16] C. R. Reeves, Ed., *Modern Heuristic Techniques for Combinatorial Problems*. USA: John Wiley & Sons, Inc., 1993.