Hardware-Aware Optimizations for Deep Learning Inference on Edge Devices

Markus Rognlien, Zhiqiang Que, Jose G. F. Coutinho, and Wayne Luk

Department of Computing, Imperial College London, 180 Queen's Gate, London, UK
jgfc@imperial.ac.uk

Abstract. AI solutions, such as Deep Learning (DL), are becoming increasingly prevalent in edge devices. Many of these applications require low latency processing of large amounts of data within a tight power budget. In this context, reconfigurable embedded devices make a compelling option. Deploying DL models to reconfigurable devices does, however, present considerable challenges. One key issue is reconciling the often large compute requirements of DL models with the limited available resources on edge devices. In this paper, we present a hardware-aware optimization strategy for deploying DL neural networks to FPGAs, which automatically identifies hardware configurations that maximize resource utilization for a given level of computation throughput. We demonstrate our optimization approach on a sample neural network containing a combination of convolutional and fully connected layers, running on a sample FPGA target device, achieving a factor of 3.5 reduction in DSP block usage without affecting throughput when using *performance* mode. When using the *compact* mode, a factor of 7.4 reduction in DSP block usage is achieved, at the cost of 1.8 times decrease in throughput. Our approach works completely automatically without the need for human intervention or domain knowledge.

Keywords: Field-programmable gate array · High-level synthesis · Meta-programming · Particle physics

1 Introduction

While traditionally reserved for compute-intensive cloud based applications, breakthroughs in hardware capabilities and efficient algorithms have led to AI solutions, such as Deep Learning (DL), becoming increasingly ubiquitous in lightweight edge devices. The clear utility of moving the implementation of DL inference as close as possible to the data source and end user has already been demonstrated with applications like wearable technology, manufacturing and agriculture, highlighting the beginnings of a significant paradigm shift (Bierzynski et al. 2021).

FPGAs make a compelling option for deploying DL models to edge devices due to their extreme parallelization capabilities as well as excellent power efficiency. However, the combination of increasingly high compute demands of modern DL models and the inherently restricted hardware resources of reconfigurable

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
L. Gan et al. (Eds.): ARC 2022, LNCS 13569, pp. 118–133, 2022.
https://doi.org/10.1007/978-3-031-19983-7_9

edge devices presents a clear challenge. If the model that one is attempting to realize in a designated reconfigurable device requires compute resources exceeding the amount available, one is left with three main options:

1. The size, and thereby computing cost, of the model can be compressed with both pruning and quantization (Duarte et al. 2018), which requires expertise and effort to prevent degrading the inference performance and accuracy;
2. The targeted device can be replaced by another device with a larger hardware budget, which leads to additional monetary cost and possibly increase in power consumption;
3. The synthesized design can be configured to share (reuse) hardware resources within each DL layer. The use of such a "reuse factor", n, for a given design, would allow for a roughly n-fold reduction in DSP block usage, but accompanied by a corresponding increase in the initiation interval (II) of the realized design (Duarte et al. 2018). In other words, with this method, resource utilisation is decreased while maintaining the original accuracy, however computation throughput would be reduced by roughly a factor of n.

While using a single universal reuse factor across all layers of a DL neural network would result in the aforementioned throughput penalty, previous works (Que et al. 2021) have demonstrated that reuse factors for each layer of a neural network can be set independently. More specifically, this work demonstrates that the careful selection of layer-wise reuse factors can be performed in order to balance the IIs of each layer of the deployed model, thereby potentially reducing the hardware requirements of the design considerably without any changes to its throughput.

Previous efforts to balance IIs using reuse factors relied on the time and attention of human programmers with domain knowledge. This work aims to entirely automate this balancing process. To the best of our knowledge, this is the first work that automatically determines a balancing configuration of reuse factors for a neural network model with heterogeneous layers on FPGAs. Our method will help accelerate the deployment of DL models in edge devices, enabling more efficient usage of limited hardware resources. In particular, this work provides the following key contributions:

C1. An approach that automatically balances the IIs of each computation stage by iteratively modifying their individual sharing levels to reduce resource utilisation while minimising performance degradation (Sect. 3);

C2. The implementation of our approach using the Artisan Metaprogramming Framework (Vandebon et al. 2021), HLS4ML (Fahim et al. 2021) and Xilinx Vivado HLS (Sect. 3.1);

C3. An evaluation of our approach under different settings for a sample neural network (Sect. 4).

2 Background

2.1 Neural Networks

Out of all methods falling under the large umbrella of Artificial Intelligence and Machine Learning, none have been nearly as influential and widely employed as the neural network, often called Deep Learning (DL). Though recent breakthroughs in the domain of DL have utilized more advanced techniques to achieve impressive results, the basic mechanisms forming the foundation of neural networks are simple. The vast majority of neural networks can be divided into *layers* (see Fig. 5), which transform one vector representation of a data sample, known as tensors, into another (possibly of different dimensions). Mathematically, these layers can be generally formalized as affine transformations (a tensor multiplication and an addition) followed by pointwise non-linearity. Simply having a sufficient number of appropriately large layers provide neural networks with the ability to approximate *any* function (Hornik et al. 1989), making them conceptually simple, yet powerful function approximators.

Fully connected (alternatively *linear* or *dense*) layers denote layers where every element in the output tensor is dependent on every element in the input tensor. *Convolutional* layers, on the other hand, found in Convolutional Neural Networks (CNNs) greatly save on memory and computation costs by allowing each element in the output tensor to only be affected by a local neighborhood of elements in the input tensor and by reusing the *weights* by which these parameters interact for the computation of each element.

From a computational standpoint, this layer-wise view of neural networks allows us to treat them like computations consisting of stages which have to be performed sequentially, but which internally are highly parallelizable.

2.2 Deep Learning on Edge Devices

Because of their typically resource-demanding nature, DL applications have traditionally been considered as confined to dedicated hardware accelerators in central cloud systems. Continued innovation in the capabilities of lightweight devices along with strategies for compressing and improving the efficiency of DL models have challenged this notion, expanding the frontier of AI applications to the edge. The prospect of enabling real-time AI inference of collected data, without the need for constant communication with a central server present obvious benefits in applications ranging form wearable health devices to autonomous vehicles.

A central challenge when deploying DL models to edge devices is ensuring sufficient compute capabilities, particularly with respect to parallelization, in order to achieve acceptable responsiveness. Another challenge is power efficiency, as both battery life and thermal considerations typically weigh heavily in their design specifications. Hence, reconfigurable devices, such as FPGAs, lend themselves particularly well to addressing these challenges, making them an exciting tool in the deployment of AI at the edge (Bierzynski et al. 2021).

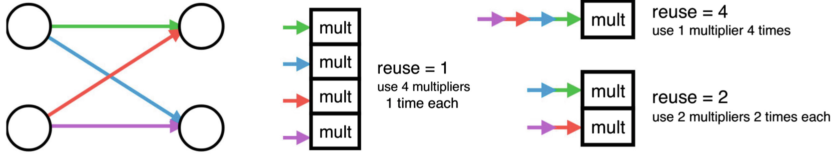


Fig. 1. (Source: https://fastmachinelearning.org/hls4ml/_images/reuse_factor_paper_fig_8.png) The figure shows how increasing the reuse factor of a computation can reduce the amount of hardware required at the cost of increased processing time.

2.3 Parallelization Optimizations Using HLS

There are many previous studies focusing on parallelization optimizations using HLS (High-Level Synthesis) tools. MPSeeker (Zhong et al. 2017), a rapid performance/area estimation framework, is proposed to explore various fine-grained and coarse-grainded parallelism options for FPGA-based accelerators at an early design stage without invoking HLS tools. The work in (Li et al. 2015) proposes resource-aware throughput optimization using HLS for multi-loops. The work (Oppermann et al. 2019) presents SkyCastle, a resource-aware multi-loop scheduler, for HLS-based kernels composed of multiple, nested loops.

Reuse factors (Duarte et al. 2018) can be used to control the number of times each computational unit should be reused in a given stage of the program, as shown in Fig. 1. For the neural network used in this work, each of the 6 layers has its own independent reuse factor which represents the number of time a multiplier is used in the computation. In the case where all reuse factors are set to 1, every single multiplication that needs to be performed over the course of the program translates to a single multiplier unit on the FPGA. In the case where the reuse factors are set to R, each multiplier unit performs R multiplications per run of the program, and only $\frac{1}{R}$ of multipliers are needed with an increased IIs. By adjusting the reuse factors of individual layers, it is possible to approximately balance the IIs of the layers, resulting in equivalent throughput at potentially much lower resource costs or a higher throughput with similar resource costs (Que et al. 2021).

3 Approach

3.1 Design Flow

Figure 2 illustrates our design-flow, which automatically translates a Deep Learning Neural Network Architecture to an optimized RTL design, which we explain next.

First, we employ the HLS4ML tool (Fahim et al. 2021) to convert the Tensorflow model to the corresponding C++ code, using 16-bit fixed point precision for both weights and activations, which can be synthesized to hardware using Xilinx

Vivado HLS. The initial C++ code version is configured to be fully parallelized, with the reuse factor for each layer set to one. Because this initial code disables resource sharing, it can potentially overmap on small FPGA devices, and thus can benefit from our optimisation approach.

Once the initial C++ code is derived, we start the feedback loop cycle and generate different C++ versions based on the original version, where we experiment with different reuse factors for each layer. We use the Artisan meta-programming framework (Vandebon et al. 2021) to manage different C++ versions, including cloning the original version, setting reuse factors based on the optimization algorithm (see Sect. 3.2), running the Vivado HLS tool on the cloned version, and interpreting the corresponding HLS reports.

The optimization algorithm is heuristic-based and iterative, keeping track of the designs that offer the best trade-off (Pareto points), while discarding all other versions. The optimization algorithm operates using two modes: *performance mode* and *compact mode*, which govern how aggressive the algorithm is with reducing the hardware resource requirements. The performance mode attempts to reduce space while preserving close to original performance, while compact mode can offer substantial savings but at a greater hit in performance. The optimization algorithm also allows to control the maximum number of iterations, which corresponds to how exhaustive the algorithm should operate in cases where it does not converge earlier.

Once the algorithm terminates, it outputs the RTL design with the optimized reuse factors for each layer. The Verilog code can be further processed to generate the bitstream to configure the FPGA device.

3.2 Optimization Algorithm

This section provides a high-level overview of how the proposed method balances IIs using reuse factors. The basic idea of the algorithm centers on maintaining piecewise linear estimates of how the IIs of the individual layers vary with their reuse factors. The method iteratively selects the most balanced configuration of reuse factors, according to these estimates, to simulate, and uses the result of this simulation to further refine its estimates. Below follows a high-level step-by-step overview of the main stages of the algorithm (see Algorithm 1).

1. A design using a reuse factor of 1 for all layers is synthesized, followed by a design using a user-specified reuse factor, λ, for all layers. The layer with the highest increase in II between these two syntheses is designated as the *anchor* layer. (Lines 1–4).
2. For a fixed anchor reuse factor (initially 1), linear interpolation is used to estimate the anchor's II at this reuse factor. This II becomes the *target* for this part of the balancing process, shown as a red line in Fig. 3. (Line 8)
3. For each other layer, the reuse factor which results in an II closest to the target based on linear interpolation is selected. This is illustrated by the dashed blue line in Fig. 3. (Lines 9–16)

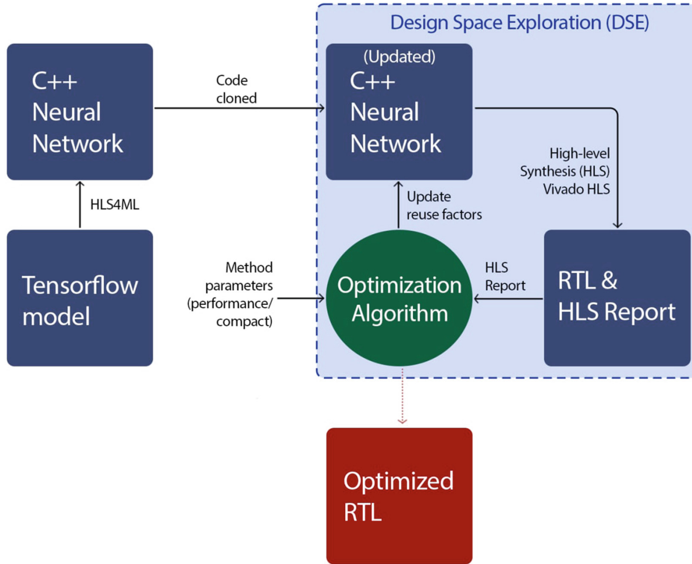


Fig. 2. The diagram illustrates the design-flow and the main feedback loop proposed in this project.

4. An error metric is then calculated as the negative sum of relative deviations from the highest estimated II. This metric is an estimate of the amount of "wasted resources" due to II bottlenecking and is shaded red in Fig. 4.
5. Steps 2–4 are repeated for every anchor reuse factor up to a user-defined maximum, M. As evident from line 7 of Algorithm 1, increasing this parameter expands the search space of our method, enabling more aggressive increases in reuse factors, leading to more hardware efficient designs at the cost of reduced throughput.
6. The configuration of reuse factors which has the minimal estimated error, as defined in step 4, is synthesized, reporting IIs for all layers. These values are used to refine the II estimators for the method's next iterations. (Lines 18, 22, 26)
7. Steps 2–6 are repeated until one of three things happen (Lines 19, 23, 28):
   (a) The configuration suggested for synthesis has been suggested before;
   (b) The set of IIs reported by the synthesis have been reported before;
   (c) The method runs for a user-specified maximum number of iterations.

Note that in addition to the specification of the M-parameter, dictating whether the method runs in performance (when $M = 1$) or compact mode (when $M \geq 2$), this algorithm relies on two user inputs which can affect the quality of the solution: λ and the maximum number of iterations. The λ parameter defines the range over which initial estimates for II gradients will be computed, and should be of the same order of magnitude as the expected optimal reuse factors. A value of $\lambda = 10$ is used in this work for this reason. The maximum number of

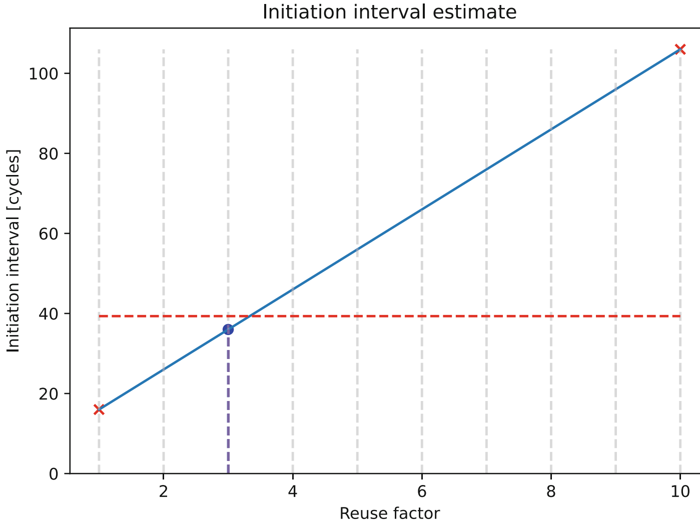


Fig. 3. An illustration of how an optimal reuse factor (according to the estimate) is selected to get as close to the target (red line) as possible. The red crosses indicate previous synthesis data for this layer and the solid blue line represents the model's estimates of IIs for all reuse factors. The dotted blue line indicates the integer reuse factor, in this case 3, which maps to the II closest to the target. (Color figure online)

iterators places an upper bound on the computation time of the algorithm, but might cause the algorithm to terminate before convergence has occurred. We used a value of 10 for this parameter, but the iteration limit was never invoked during any of our evaluation.

4 Evaluation

4.1 Experimental Setup

In order to evaluate the performance of our method, we employ a neural network designed to be deployed across numerous edge devices in the processing pipeline of the Large Hadron Collider. This particular network, shown in Fig. 5, classifies fundamental particles based on their jet substructure, and contains a combination of convolutional and fully connected layers of various sizes, allowing us to evaluate our method's performance in a heterogeneous environment. The network was specified and trained on the same dataset (training (60%), validation (20%), and testing (20%)) as (Duarte et al. 2018) using Tensorflow 2.91, and converted to a C++ DL kernel using HLS4ML 0.6.0. This work splits the whole model into several layers and utilizes a layer-wise hardware architecture (Duarte et al. 2018; Tridgell et al. 2019; Nakahara, 2020) which maps all the layers on-chip. It is flexible and able to take full advantage of the customizability of FPGAs. In addition, the design implements a coarse-grained pipeline to further increase the design throughput.

Algorithm 1. Balancing Optimization Algorithm

```
Input:
    k                        ▷ C++ DL kernel with N layers
    M                        ▷ resource efficiency level (high = more efficient)
    λ                        ▷ initiation parameter
    max_iterations           ▷ maximum number of iterations
Output:
    C                        ▷ optimized reuse factors for each layer [r_1, r_2, ..., r_N]
 1: D ← ∅
 2: D ← D ∪ hls_synth(k, [1, 1, 1..., 1])
 3: D ← D ∪ hls_synth(k, [λ, λ, λ, ..., λ])
 4: anchor ← argmax{D[i, λ] − D[i, 1], i ∈ 1..N}
 5: iter ← 0
 6: repeat
 7:     for m ← 1..M do
 8:         ii_target ← ln_interpolation(D[anchor, :], m)
 9:         for layer ← 1..N do
10:             if layer ≠ anchor then
11:                 E[m, layer] ← argmin{
                        |ii_target − ln_interpolation(D[layer, :], r)|,
                        r ∈ N
12:                 }
13:             else
14:                 E[m, layer] ← m
15:             end if
16:         end for
17:     end for
18:     C ← argmin{error_estimate(E[m, :]), m ∈ 1..M}
19:     if previously_synthed(C) then
20:         return C
21:     end if
22:     result ← hls_synth(k, C)
23:     if result ∈ D then
24:         return C
25:     end if
26:     D ← D ∪ result
27:     iter ← iter + 1
28: until iter == max_iterations
29: return C
```

This project was then synthesized, with reuse factors for all layers set to 1, forming an unoptimized baseline for later comparison. Our method was then applied, utilizing Artisan 1.0.7 and Vivado HLS 2019.2 to produce optimized designs for two different parameter settings: *performance* and *compact*, respectively. As previously mentioned, the performance mode ($M = 1$) aims to reduce the hardware resource requirements with only zero to minor reductions in throughput, while the compact mode ($M \geq 2$), works more aggressively to

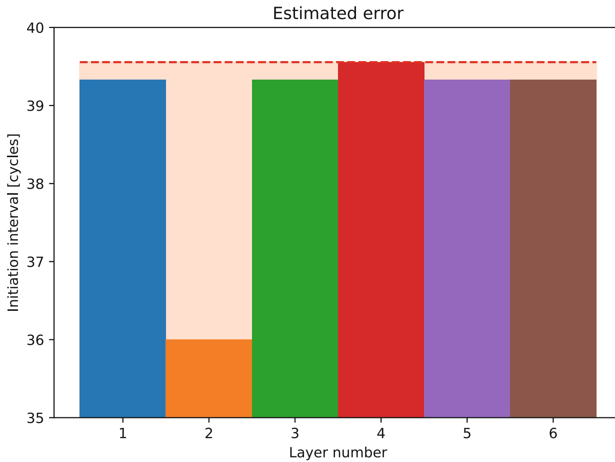


Fig. 4. An illustration of how the estimated error for a given configuration of reuse factors is computed. The IIs corresponding to the reuse factors are estimated from interpolation, and the un-normalized error is then found by computing the negative deviations from the highest of these II estimates, shown here as the area shaded light red. The y-axis has been truncated to highlight this process. (Color figure online)

reduce DSP block usage at the expense of performance. The second synthesis parameter, λ was set to 10 for both runs. We evaluate both settings in the following subsections.

4.2 Performance Mode

By setting the M-parameter of the optimization algorithm to 1, the search space of valid reuse factor configurations is restricted to the set for which the lowest reuse factor of any layer is 1. This ensures that the total II of the full network does not drastically increase, if at all.

After synthesizing the two cases of all reuse factors being 1 and all being 10, the method converges after two iterations of the algorithm in performance mode. The initial synthesis where all reuse factors are 1 as well as the two configurations suggested by the algorithm are shown in Fig. 6. The synthesis where all reuse factors were set to 10 is excluded from this and similar plots, as adjusting the scale of the y-axis to fit the results of this simulation would make it difficult to discern differences in the remaining groups of bars.

The proposed method came very close to finding the optimal configuration with its first suggestion, setting every reuse factor to its optimal value apart from that of layer 5. Examining why this happened highlights a key aspect of how the algorithm works. Looking at Fig. 7, we can see the algorithm's estimate of how the II of layer 5 varies with respect to its reuse factor both before and after synthesizing its first suggestion. The algorithm under-estimates the II of layer

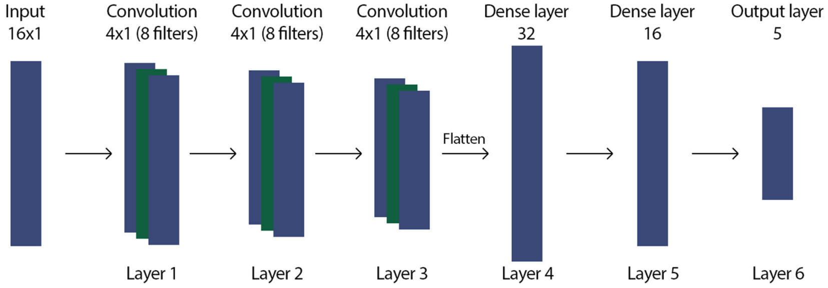


Fig. 5. An overview of the structure of the neural network optimized and synthesized for an FPGA to evaluate our approach. The network consists of 6 total layers, 3 1D-convolutions followed by 3 linear layers of shrinking size. The activation function following the first five layers is ReLU, and since the network aims to solve a classification problem, the output activation function used is SoftMax.

5, but uses the information it gains from the penultimate synthesis to arrive at the optimal reuse factor.

4.3 Compact Mode

If the balanced solution obtained with *performance* mode does not fit the target device and a decrease in throughput can be tolerated, the value of M can be increased ($M \geq 2$) to enable the *compact* mode which explores configurations with larger reuse factors.

For instance, setting M to 2 expands the search space to configurations where the smallest reuse factor of any layer is no larger than 2. In this case, the algorithm makes a total of 5 suggestions before converging. These suggestions, in addition to the initial synthesis of all reuse factors set to 1, can be seen in Fig. 8.

While the algorithm using the *performance* mode terminates due to the algorithm suggesting a configuration of reuse factors which has been previously suggested and synthesized, the *compact* mode terminates because two consecutive syntheses yield identical IIs for all layers. This early stopping criterion was implemented in order to prevent fruitless gradual decrements of a single reuse factor when the algorithm believes it is close to an optimal solution, when in reality it already has one. Figure 9 illustrates how the II of layer 6 varies with its reuse factor. Were it not for this early stopping criterion, the algorithm would continue performing these marginally tweaked and expensive syntheses until a major drop to an II of 24 cycles would occur at a reuse factor of 40, proving that an II of 35 was indeed the best II in this particular instance.

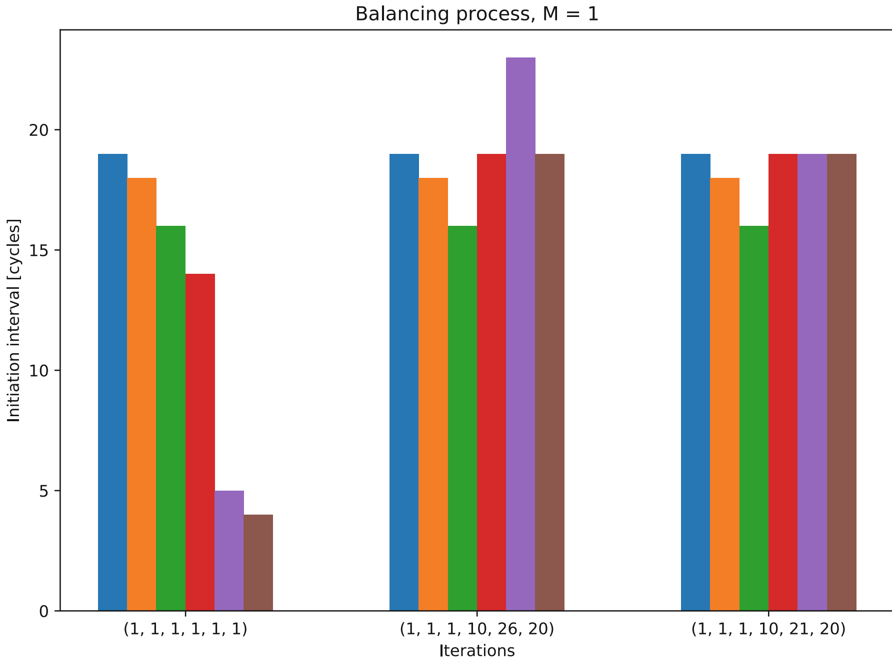
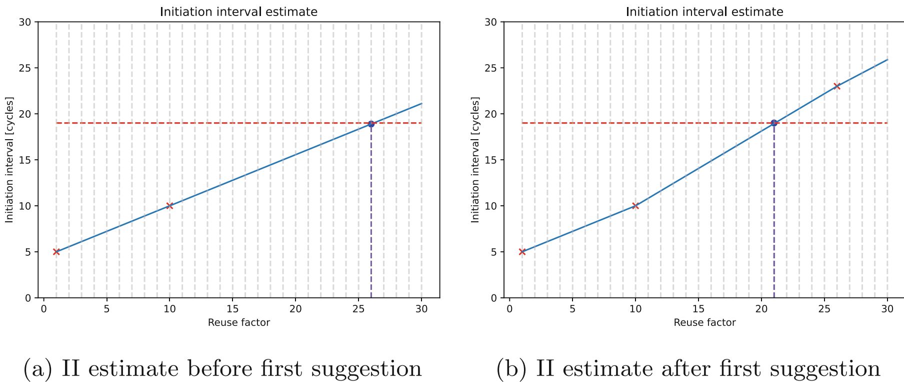


Fig. 6. An overview of the configurations of reuse factors suggested by the *performance* mode algorithm along with the base case of all reuse factors set to 1. Each of the three groups of 6 bars represent a single synthesis. The heights of the bars correspond each layer's II using the configuration of reuse factors in brackets below each group of bars.



(a) II estimate before first suggestion (b) II estimate after first suggestion

Fig. 7. These figures illustrate how the model uses information from simulations to improve its estimates and give better suggestions. The first plot shows how the model estimates the relationship between the II and reuse factor of layer 5 based on the two initial simulations. The model under-estimates the gradient when extrapolating from a reuse factor of 10, and suggests a reuse factor of 26. When the synthesis result informs the model of its under-estimate, it finds the optimal reuse factor.

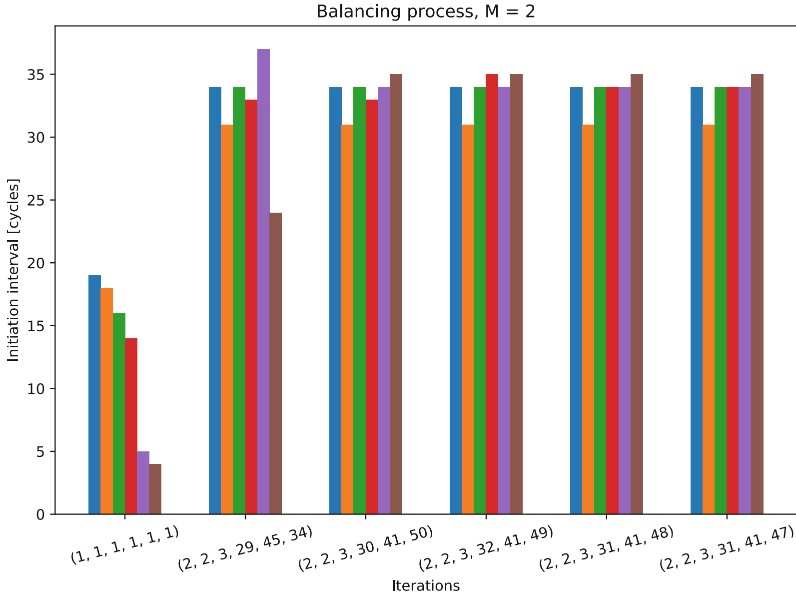


Fig. 8. Configurations of reuse factors suggested by the algorithm in *compact* mode. Each of the groups of 6 bars represent a single synthesis. The heights correspond to the IIs using the reuse factors in brackets below each group.

4.4 Comparison Between Performance and Compact Modes

Table 1 shows II and latency, as well as DSP block, FF and LUT usage, for the unoptimized design as well as designs optimized by the algorithm running in both performance and compact mode. By looking specifically at the number of DSP blocks used by each of the designs, we can see that the sample network studied in this work would not fit on the sample target device, requiring more than three times the available DSP blocks. By applying the performance mode of the method outlined in this work, the DSP block usage was reduced by a factor of 3.5. This allows the network to fit the target device without any decrease to the throughput compared with the original design, as is evident from the identical II. Besides minor increases in FF and LUT resource usage, the balancing of IIs by increasing reuse factors inevitably incurs an increase in latency.

If we wanted to realize the sample network shown in Fig. 5 in an FPGA with even fewer available DSP blocks, such that the performance mode design would not fit, we could apply the more aggressive compact mode of the method. In this case, the DSP block usage was further decreased by a factor of 2.1. This, however, forces the method to further increase both the latency and the II.

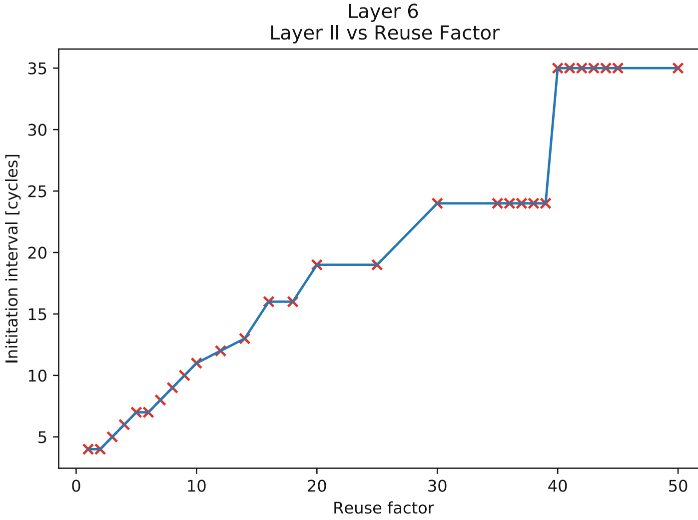


Fig. 9. This figure shows how the II of layer 6 in the sample network varies with its reuse factor. Each red cross represents an actual synthesis simulation (performed independently of any execution mode of the balancing algorithm). (Color figure online)

Table 1. Performance and resource usage comparison between the unoptimized and optimised versions automatically derived by our approach (performance and compact modes, respectively). The rightmost column represents the case where all reuse factors are set to 4 (discussed in Sect. 4.5). Resource usage percentage in brackets corresponds to the proportion of available resources used (*Xilinx Artix A200*).

	Unoptimized	Performance	Compact	Homogeneous <4>
II	20	20	36	67
DSPs	2,269 (310%)	642 (87%)	308 (42%)	737 (100%)
Latency	45	88	150	87
FFs	45,954 (17%)	51,398 (19%)	52,729 (20%)	49,387 (18%)
LUTs	100,525 (75%)	108,248 (80%)	108,291 (80%)	118,586 (88%)

4.5 Comparison with State-of-the-Art

Duarte et al. (2018) uses reuse factors to trade increased IIs, and thus decreased throughput, for reduced hardware requirements when deploying a neural network to an FPGA. Unlike our approach, which allows the reuse factor of each layer of the network to be set independently, Duarte et al. (2018) uses a single homogeneous reuse factors for all layers. If this restriction of a singular reuse factor were imposed on the sample network used for evaluation in this paper, a reuse factor of 4 would need to be applied to all layers in order to fit within the resource constraints of our chosen target device. As shown in the rightmost

column of Table 1, this would lead to a factor 3.4 decrease in throughput when compared to the design produced by our method in addition to requiring more DSP blocks.

Que et al. (2021) balanced the IIs of the layers of a Recurrent Neural Network (RNN) by modifying the reuse factors of each layer independently. However, this was done manually using expert knowledge about the tools being used, the specific properties of the network being synthesized and the target device. Without this knowledge, a naive strategy for finding an optimal configuration of reuse factor would be to synthesize designs using a singular reuse factor, n, across all layers in some plausible range $1 \leq n \leq N$, and then identifying an optimal configuration by setting the reuse factor of each layer based on the results of this simulation. As the highest reuse factor explored by our method in this evaluation is 50, covering this design space with a brute force approach would require 50 such syntheses. In our evaluation, each synthesis of a relatively small neural network takes approximately 10 min, thus the synthesis alone would take more than 8 h of compute time to derive the optimal solution with the naive method. Our method in its performance mode required only 4 such syntheses, rather than 50, while the compact mode, navigating a larger search space, required only 7. As the computation time spent by the algorithm itself (that is, time not spent on synthesis) is on the order of milliseconds and therefore negligible in this context, our method achieved a factor 7–13 speedup of the DSE process compared to a naive approach, enabling greatly accelerated iteration.

4.6 Further Work

As our method exclusively extracts II data from the HLS report, it is not able to fully exploit all available DSP blocks on the target device. A future extension to the method could use additional available data, such as latency and resource usage, to produce the configuration of reuse factors for a given DL model which maximizes throughput given the resource constraints imposed by the target device and a latency constraint manually specified by the user.

As explained in Sect. 3.2, we require a user-specified reuse factor, λ, to obtain an initial estimate of the gradient of each layer's II with respect to its reuse factor. It is possible that a heterogeneous configuration of reuse factors could provide more useful information. One extension might therefore be to compute such a configuration using the IIs resulting from the first synthesis, which should give some information about suitable orders of magnitude of the reuse factors in the final solution.

Though our method's performance has only been evaluated for one sample network, the heterogeneous nature of this network suggests that the method generalizes well across different types and sizes of neural network layers. Seeing as our method makes no assumptions about the implementations of these layers, other than that they are highly parallelizable stages of a computation, further work could be done to evaluate our method on the synthesis of other non-DL programs consisting of similarly parallelizable stages.

Finally, another method for addressing a resource-constrained environment is to perform quantization. Further work will examine how our II-balancing method can be combined with quantization schemes for further hardware reductions.

5 Conclusion

This paper has outlined a method for automatically balancing the initiation intervals (IIs) of sequential stages of a computation to iteratively modify reuse factors for each stage. For one particular algorithm setting, this method reduced the computational hardware requirement to realize a neural network by a factor of 3.5 without affecting throughput. It was demonstrated that the use of this method allowed the neural network to fit within the hardware resources of a designated target device, which otherwise would have required more than three times the available resources. If one wanted to target an even smaller device, a more aggressive parameter setting of the algorithm was shown to decrease the DSP block requirement by a factor of 7.4, while II increased by a factor of 1.8. In contrast to previous efforts to balance IIs using reuse factors, the balancing algorithm proposed in this paper is completely automatic.

In contrast to previous related works using one reuse factor for all layers, the use of multiple reuse factors in our method facilitated a factor of 3.4 increase in throughput from the same resources. Our approach is automatic and can achieve a factor of 7.5–13 speed improvement over a naive exploration of the same search space of reuse factors.

Acknowledgements. The support of the UK EPSRC (grant number EP/V028251/1, EP/L016796/1, EP/S030069/1 and EP/N031768/1) and AMD is gratefully acknowledged.

References

Bierzynski, K., et al.: AI at the edge. 2021 EPoSS White Paper (2021)

Duarte, J., et al.: Fast inference of deep neural networks in FPGAs for particle physics. J. Instr. **13**(07), P07027 (2018)

Fahim, F., et al.: hls4ml: an open-source codesign workow to empower scientifc low-power machine learning devices. arXiv (2021)

Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**, 359–366 (1989)

Li, P., et al.: Resource-aware throughput optimization for high-level synthesis. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 200–209 (2015)

Nakahara, H., et al.: High-throughput convolutional neural network on an FPGA by customized JPEG compression. In: IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE (2020)

Oppermann, J., et al.: SkyCastle: a resource-aware multi-loop scheduler for high-level synthesis. In: 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 36–44. IEEE (2019)

Que, Z., et al.: Accelerating recurrent neural networks for gravitational wave experiments. arXiv (2021)

Tridgell, S., et al.: Unrolling ternary neural networks. ACM Trans. Recongurable Technol. Syst. (TRETS) **12**(4), 1–23 (2019)

Vandebon, J., et al.: Enhancing high-level synthesis using a meta-programming approach. IEEE (2021)

Zhong, G., et al.: Design space exploration of FPGA-based accelerators with multi-level parallelism. In: 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1141–1146. IEEE (2017)