







# Low Cost Implementation of Deep Neural Network on Hardware

Gaurav Kumar<sup>(✉)</sup>, Anuj Kumar, Satyadev Ahlawat,  
and Yamuna Prasad

Indian Institute of Technology Jammu, Jammu, India  
{gaurav.kumar, anuj, satyadev.ahlawat, yamuna.prasad}@iitjammu.ac.in

**Abstract.** Recently, deep learning framework gained extreme importance in various domains such as Computer Vision, Natural Language Processing, Bioinformatics, etc. The general architecture of deep learning framework is very complex that includes various tunable hyper-parameters and millions/billions of learnable weight parameters. In many of these Deep Neural Network (DNN) models, a single forward pass requires billions of operations such as multiplication, addition, comparison and exponentiation. Thus, it requires large computation time and dissipates huge amount of power even at the inference/prediction phase. Due to the success of DNN models in many application domains, the area and power efficient hardware implementations of DNNs in resource constraint systems have recently become highly desirable. To ensure the programmable flexibility and shorten the development period, field-programmable gate array (FPGA) is suitable for implementing the DNN models. However, the limited bandwidth and low on-chip memory storage of FPGA are the bottlenecks for deploying DNN on these FPGAs for inferencing.

In this paper, Binary Particle Swarm Optimization (PSO) based approach is presented to reduce the hardware cost in terms of memory and power consumption. The number of weight parameters of the model and floating point units are reduced without any degradation in the generalization accuracy. It is observed that 85% of the weight parameters are reduced with 1% loss in accuracy.

**Keywords:** Deep neural network · FPGA · Optimization · PSO

## 1 Introduction

Deep learning techniques on hardware have gained popularity because of their good accuracy and use in wide range of applications. These techniques are used in many modern machine learning applications such as driver assistance systems, speech recognition, natural language processing, healthcare etc. DNNs are being used in various applications because of their self-adaptive features, non-linear characteristics and their ability to adapt versatile configurations [6, 9, 12]. Furthermore, DNNs learn the input data pattern layer by layer and then extract

the features. Various problems such as face recognition, medical image segmentation and classification, action recognition/classification, autonomous vehicle driving and machine translations have been successfully solved using deep learning frameworks. The top leading industries such as Google, Microsoft, IBM and Amazon etc. are exploiting and enhancing these deep learning models for analyzing the massive data collected from various social media sources.

As the research moves toward success in terms of accuracy, there is a trade-off between computational time and model complexity. To reduce the computational time, research is facilitated towards the efficient implementation of these models on hardware. Since deep learning techniques need to extract a large number of features from raw data, it requires huge computational resources and hence complex hardware to perform operations such as addition and multiplication on large matrices. Thus, as the models get complex, it becomes difficult to implement these DNN applications in resource constraint devices. Also, various parameters such as cost, efficiency, power consumption and accuracy will influence the hardware implementation of these deep neural networks, especially for resource constraint applications. In the learning phase, these DNNs learn millions/billions of parameters which are stored in the memory. On the other hand, inference is a compute-intensive task with a large number of floating-point operations. The limited on-chip memory is a considerable bottleneck in storing these millions/billions of parameters and the low power budget becomes a huddle in inference. In order to exploit the FPGAs in real-time inferencing, the following three approaches are popular in the literature [11, 18, 20]:

1. Reducing the precision of floating point units.
2. Parameter reduction and use of approximate computing architectures.
3. Partitioning of the problem to implement on multiple FPGAs.

In this work, a Particle Swarm Optimization (PSO) based approach is proposed to decrease the memory requirement and power consumption. The proposed mechanism reduces the hardware cost by reducing the weights of the model using a stochastic optimization technique. This results in alleviation of the precision of floating point units with insignificant loss in the accuracy. Our contributions in this study are as follows:

1. The memory used to store the model weights is reduced by parameter pruning. The parameters of trained DNN Models are pruned using evolutionary algorithms without much loss in generalization accuracy.
2. The power consumption in computation is reduced by low precision transformation on the pruned model parameters.
3. A generalized mechanism is created to reduce the number of weights of any heavy DNN models such as AlexNet, DenseNet, VGG16, etc.

## 2 Background

In the literature, evolutionary algorithm based approaches have been widely studied for multi objective optimizations and parameter tuning [7, 19]. These

approaches look for an approximate solution by choosing an initial population of candidate vectors with random values and refining the solution based on the multi-objective fitness criteria [19]. These approaches could be explored to reduce the number of parameters in the trained deep learning models.

## 2.1 Formulating the Optimization Problem

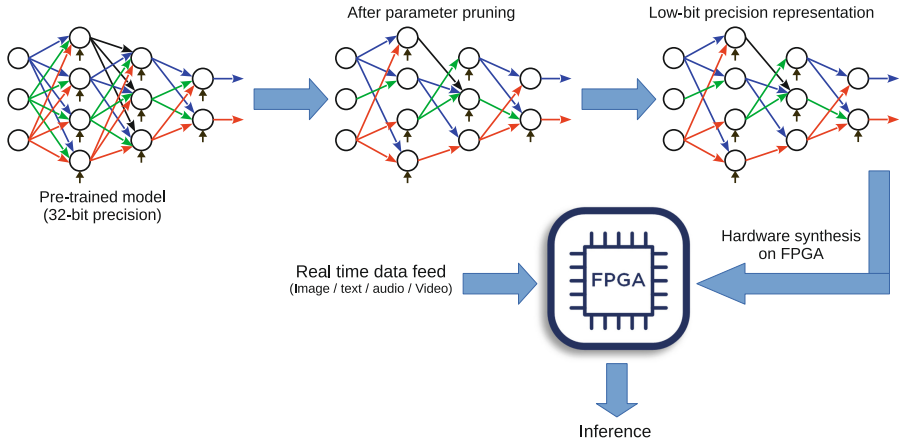
The heuristic algorithms such as Genetic Algorithms, Differential Evolution [16] and Particle Swarm Optimization (PSO) are widely used in non-convex and multi-objective optimization problems. These methods primarily require a fitness function whose value (fitness value) is calculated for a given set of decision variables. Before the beginning of the optimization procedure in the algorithm, a population is randomly generated. This population is termed as a collection of chromosomes/particles/agents/individuals in the framework of heuristic algorithms. As the iterations of the algorithm proceed, the members of the population are modified because of the evolutionary approach being followed. At the end of iterations, the particle or chromosome that gives the optimized value for the fitness function is selected. Depending upon the kind of optimization (maximization/minimization) being done, either highest value or lowest value is selected as the optimized value and thereby the particle or chromosome which gives the optimized value is taken as the solution.

To fit the problem of parameter estimation of the Deep Learning model into this framework, a fitness function and decision variables are required. The purpose of the fitness function is to indicate how good are the solutions given by the heuristic algorithm. Furthermore, the decision variables' values influence the fitness function's value. However, it might be possible that the best values require a large number of iterations. Thus, to reduce the time, the termination criteria for the algorithm is the number of iterations.

## 2.2 Deep Learning for Hardware

In recent years, various types of dedicated hardware have been developed that target deep learning techniques. For example, in order to perform multiple floating-point operations with quick processing and high precision, a 16-bit floating-point arithmetic is being used in Nvidia Tesla P100. Also, Big Basin is a Facebook deep neural network server that performs DNN processing efficiently [13].

Deep neural network models could be implemented on various hardware platforms such as FPGAs, ASICs and SOCs [10,14]. Although the DNN implementation on CPU is flexible as well as cost-efficient, it is performance constrained. On the other hand, DNN implementation on ASIC results in better performance than CPU. The DNN implementation on SOCs is also gaining popularity due to its ability to provide both hardware as well as software processing ability in a single device. Hardware could be used as an accelerator, whereas software is coded to perform a specific job/task.



**Fig. 1.** The overall architecture of proposed approach.

### 3 Methodology

The proposed approach is shown in Fig. 1, which consists of basic DNN, an optimization algorithm to reduce the number of weights and FPGA implementation of the reduced model.

#### 3.1 Deep Neural Network

To train the DNN model, MNIST [1] dataset has been taken as input, which is a hand digit recognition dataset. The DNN model represented as  $F(X)$  have input  $X$  where,  $X \in (x_1, x_2, x_3, \dots, x_n)$ , and output  $Y$  where,  $Y \in (y_0, y_2, y_3, \dots, y_m)$ . The practical DNNs have hundreds or thousands of layers with thousands of neurons in each layer, making total parameters in millions or billions. These parameters need to be stored in the memory for inferencing phase.

There are many pre-trained models available in well known TensorFlow library<sup>1</sup>. A few of them are briefly explained below:

1. **AlexNet [8]:** Standard AlexNet consists of 8 layers, out of which 5 are convolution layers and 3 are fully connected layers. It is trained on the ImageNet dataset. AlexNet revolutionized the state-of-the-art in object recognition in the real time.
2. **DenseNet [5]:** This model is similar to a dense layer feed-forward network where each layer is fully connected with the successive layer. This model has been primarily used for object recognition tasks on ImageNet, CIFAR, SVHN and CIFAR100 datasets.
3. **VGG16 [15]:** This model has been used for ImageNet dataset classification and is also widely used as a transfer learning pre-trained model in low resource domains.

<sup>1</sup> <https://keras.io/api/applications/>.

In our experiments, AlexNet [8] is chosen to demonstrate the effectiveness of the proposed approach. The chosen model is a moderate-size DNN model suitable for the proof of the concept.

### 3.2 Particle Swarm Optimization (PSO)

To optimize the number of parameters, an existing optimization algorithm, i.e., Particle Swarm Optimization (PSO) algorithm [7] is used. This is a bio-inspired algorithm which does not require a convex objective function. This algorithm works for any non-convex, discrete and multi-objective function optimization scenario. In most cases, the objective function serves as part of the fitness function. The fitness of a particle/position/individual represents the goodness of the solutions. A few hyper-parameters are required for the PSO algorithm. The fitness function for the minimization is

$$f = \alpha \times (1 - accuracy) + (1 - \alpha) \times \frac{count_x(1)}{total\_features(x)} \quad (1)$$

Here, accuracy is the model accuracy after parameter pruning by particle swarm optimization (PSO) algorithm, and  $count_x(1)$  is the remaining number of parameters after pruning. The proposed non-convex fitness function jointly minimizes the validation error and number of parameters. The parameter  $\alpha$  controls the contribution of model validation error and the number of parameters in the joint optimization.

The PSO algorithm tries to minimize the fitness function so that it can increase the accuracy as well as decrease the number of parameters. It tries to get the best position for each particle by updating its local best ( $pbest$ ) and then global best ( $gbest$ ) position. In this work, the position vector is represented using a binary vector of '1s' and '0s'. The value '1' represents the inclusion/selection of the corresponding weight parameters, while the value '0' represents the rejection of the corresponding weight parameters in the model. Thus, an optimized position vector represents the optimized number of parameters. Initial velocities and positions are randomly generated for each particle and then updated according to the following equation:

$$v_{(i+1)} = w \times v_i + c_1 \times r_1 \times (pbest_i - x_i) + c_2 \times r_2 \times (gbest - x_i) \quad (2)$$

where  $v_i$  is the velocity of  $i^{th}$  particle,  $x_i$  is the position of  $i^{th}$  particle,  $w \in (-1,1)$ ,  $c_1 + c_2 \leq 4$ , and  $r_1, r_2 \in (0,1)$  which are randomly generated

$$x_{i+1} = \begin{cases} 1 & \text{if } \text{sigmoid}(v_{i+1}) \geq 0.5 \\ 0 & \text{else} \end{cases} \quad (3)$$

The PSO Algorithm 1 follows the following steps to find out the optimum solution:

1. In first step, random positions and random velocities for each particle are collected. Each particle has its own *pbest* position and each iteration has a *gbest* position.
2. In second step, the fitness function is calculated on the positions of all the particles and again choose the *pbest* and *gbest*.
3. In next step, the algorithm re-calculates the velocities of each particle with local best and global best positions.
4. Then, with the help of new velocities, it re-calculates the optimum positions, i.e., local best of each particle. Because of binary PSO, rather than simply adding the position vector with velocity, it uses another method explained in next step.
5. Finally, to update the position vector, sigmoid of velocities are taken and if they are greater than 0.5, then the position vector has a value of 1; else 0.
6. The above steps are repeated until a near-optimum solution is observed or some loop-breaking condition is met.

**Complexity:** The complexity of the Algorithm 1 is  $\mathcal{O}(T.k.m.F)$ ; where  $T$  denotes the number of maximum iterations,  $k$  denotes the number of particles/individuals,  $m$  denotes the number of parameters and  $F$  denotes the complexity of fitness function.

---

**Algorithm 1.** Binary Particle Swarm Optimization

---

```

1: Input: Fitness Function, Initial Positions( $x_i$ ), Initial Velocities( $v_i$ )
2: Output: Optimal Solution
3: for each particle (i) do
4:   Calculate fitness function ( $f_i$ )
5:   Update  $pbest_i$  and  $gbest$ 
6: end for
7: while iteration do
8:   for each particle (i) do
9:     Update velocities ( $v_{(i+1)}$ ) and positions ( $x_{(i+1)}$ )
10:    Update  $pbest_{(i+1)}$  and  $gbest$ 
11:   end for
12: end while
13: Return: Optimal Solution

```

---

### 3.3 Low Bit Precision Representation

For the floating point representation of real numbers, the IEEE 754 standard is used. It has three components: sign, exponent and fraction. The sign is of 1 bit, exponent is of 8 bit and fraction is of 23 bit, which makes a word size of 32 bits. The reduction in the number of bits for floating point representation requires less complex hardware and hence reduces the power consumption. Floating point is a quantization of infinite precise real numbers. The standard floating point

representation can exactly represent the real number in the range of  $10^{-5}$  to  $10^5$ . If the number is out of this range then, it is rounded off. However, most deep neural networks perform their calculations in a relatively small range, such as  $-10.0$  to  $10.0$ . Thus, by compressing the 32-bit weight matrix into 8 or 16 bits after parameter pruning, there is no significant loss in the accuracy. The experiments show that 1 bit for sign, 5 bit for exponent and 2 bit for fraction in case of 8 bit representation and 1 bit for sign, 11 bit for exponent and 4 bit for fraction in case of 16 bit representation produce best results.

## 4 Experiments

To evaluate the effectiveness of the proposed approach, extensive experiments are performed. The training of deep learning models involves various matrix operations such as multiplication (MACC), addition (ADD), comparison (COMP), division (DIV) and exponentiation. In many DNN models, billions of operations are required, as shown in Table 1. Thus it requires high power budget and computation time. The Graphics Processing Units (GPUs) were designed to exploit SIMD architecture. The GPUs provide more computational power with a fast connection to memory. The GPU architectures are widely used to train and inference large deep layered neural networks such as VGG16 [15], Inception V3 [17], ResNet-152 [4] and BERT [2] etc. The GPU devices are very power-hungry, which limits their usage for real-time applications using mobile devices, IoTs etc.

As shown in Table 1, AlexNet has 60.97 M parameters which are trained and stored in the memory for inferencing. Also, it has 17.86 M parameters in convolution layers. We use AlexNet for our experiments as it has significantly large number of parameters.

### 4.1 Results

The PSO algorithm optimizes the number of parameters without any significant loss in generalization accuracy. As can be observed from Fig. 2, the fitness func-

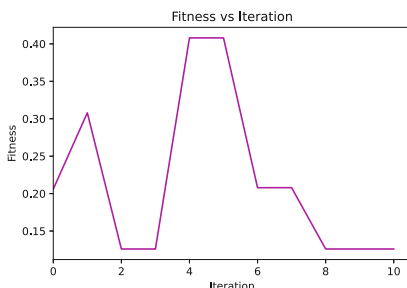
**Table 1.** Deep learning model details [3]

Model	MACC	COMP	ADD	DIV	Activations	Params
InceptionV4	12.27 G	21.87 M	53.42 M	15.09 M	72.56 M	42.71 M
InceptionV3	5.72 G	16.53 M	25.94 M	8.97 M	41.33 M	23.83 M
InceptionV2	13.18 G	31.57 M	38.81 M	25.06 M	117.8 M	55.97 M
ResNet-152	11.3 G	22.33 M	35.27 M	22.03 M	100.11 M	60.19 M
ResNet-50	3.87 G	10.89 M	16.21 M	10.59 M	46.72 M	25.56 M
AlexNet	7.27 G	17.69 M	4.78 M	9.55 M	20.81 M	60.97 M
GoogleNet	16.04 G	161.07 M	8.83 M	16.64 M	102.19 M	7 M
VGG16	154.7 G	196.85 M	10 K	10 K	288.03 M	138.36 M
BERT -12	—	—	—	—	—	110 M
BERT-24	—	—	—	—	—	345 M

**Table 2.** Parameter values for PSO

Name	Value of the parameter
Number of particles	10
Maximum iteration	100
alpha( $\alpha$ )	0.7
omega( $w$ )	0.99
$c_1$	2.0
$c_2$	2.0

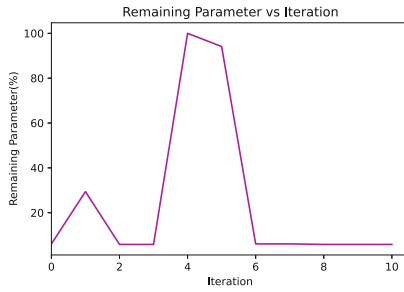
tion is minimized as the number of iterations increases. The inertia weight  $w$  and velocity vector are initialized with 1.0 and 0.1, respectively, in the experiments. After every iteration,  $w$  is multiplied by 0.99. Table 2 provides the details of other parameters. It should be noted that there are some rises and falls due to the local minima. As shown in Fig. 2, the fitness value saturates after 8<sup>th</sup> iteration. There is no further reduction which denotes the completion of the algorithm. Now, the reduced number of parameters is calculated using the *gbest* position vector. The parameters after parameter pruning are only 15% of the total parameters with 1% loss in accuracy (i.e., 98.20%, Table 3). It has also been observed that proposed optimization heuristic achieves 50% reduction in the number of parameters without any degradation in the validation accuracy (Table 3). Additionally, a comparison of the proposed PSO based approach with the Random Parameter selection method (RP) is provided in Table 3. The RP method achieves a 50% reduction in total parameters with 8% loss in accuracy.



**Fig. 2.** Fitness vs Iteration

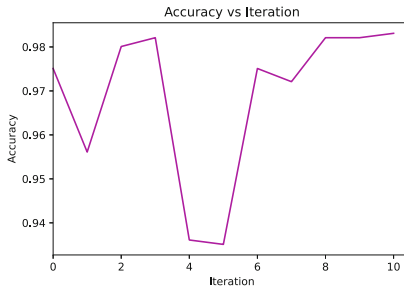
The parameters count after each iteration of the algorithm is shown in Fig. 3. It is evident from Figs. 2 and 3 that as the number of parameters increases, the fitness value also increases. The accuracy of the model after each iteration of parameter pruning by the PSO algorithm is shown in Fig. 4. The proposed approach achieves a similar set of accuracy with 85% reduction in the number of parameters.





**Fig. 3.** Remaining Parameters vs Iteration

The original model and model after parameter pruning (proposed model) with different floating-point precision are compared in Table 4. It can be observed that the proposed model reduces the model size by 50% while producing a similar set of accuracy. It should be noted that the accuracy degrades when the floating point precision is reduced (Table 4).



**Fig. 4.** Accuracy vs Iteration

### 4.2 Hardware Cost

By using PSO algorithm, 85% of the total parameters are reduced, which directly results in the reduction of total required memory. A further saving of the memory (by a factor of two) can be achieved using low bit representation for the remaining 15% parameters. Moreover, the reduction in the total parameters directly reduces the required power budget. This is because the energy required for the floating point multiplication reduces significantly. Instead of a floating point multiplication, now it becomes an integer multiplication with zero.

**Table 3.** Model validation accuracies for MNIST dataset

Method	#Parameters	Accuracy
Original Model (AlexNet)	17,866,752	99.40
Random parameter selection	9,930,946	91.65
Proposed Model (without any loss)	<b>8,933,370</b>	<b>99.40</b>
Proposed Model (with loss)	<b>2,680,012</b>	<b>98.20</b>

**Table 4.** Model Sizes with different floating point precision and accuracy

Name	64-bit float		32-bit float		16-bit float	
	Size (in MB)	ACC	Size (in MB)	ACC	Size (in MB)	ACC
Original model	142.934	99.40	71.467	98.20	35.733	97.50
Proposed model	69.467	98.20	33.733	97.80	17.733	96.80

## 5 Conclusion and Future Work

In this paper, a generalized mechanism to reduce the cost (in terms of area and power) of a hardware implementation of Deep Neural Network is proposed. It has been shown that 85% of parameters could be reduced with 1% loss of accuracy with parameter pruning. A further reduction in required memory space is achieved with low bit floating point representation for remaining parameters. Thus, making it suitable for resource constrained embedded devices. For future work, the more advanced and complex deep neural network will be implemented on hardware using this scheme.

**Acknowledgements.** This work has been partially supported by the SERB DST under project number SRG/2020/002551 and IHUB NTIHAC FOUNDATION under project numbers IHUB-NTIHAC/2021/01/14, IHUB-NTIHAC/2021/01/15.

## References

1. Deng, L.: The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Process. Mag.* **29**(6), 141–142 (2012)
2. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding (2018). <https://doi.org/10.48550/ARXIV.1810.04805>
3. GitHub: Details of deep learning models. <http://dgschwend.github.io/netscope/quickstart.html>
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition (2015). <https://doi.org/10.48550/ARXIV.1512.03385>
5. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks (2018)

6. Kamnitsas, K., et al.: Efficient multi-scale 3D CNN with fully connected crf for accurate brain lesion segmentation. *Med. Image Anal.* **36**, 61–78 (2017). <https://doi.org/10.1016/j.media.2016.10.004>
7. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of ICNN'95 - International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995). <https://doi.org/10.1109/ICNN.1995.488968>
8. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (eds.) *Advances in Neural Information Processing Systems*, vol. 25. Curran Associates, Inc. (2012)
9. Litjens, G., et al.: A survey on deep learning in medical image analysis. *Med. Image Anal.* **42**, 60–88 (2017). <https://doi.org/10.1016/j.media.2017.07.005>
10. Lu, L., Liang, Y., Xiao, Q., Yan, S.: Evaluating fast algorithms for convolutional neural networks on FPGAs. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 101–108 (2017). <https://doi.org/10.1109/FCCM.2017.64>
11. Nakahara, H., Zhiqiang, Q., Jinguji, A., Luk, W.: R2CNN: recurrent residual convolutional neural network on FPGA. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2020, p. 319. Association for Computing Machinery, New York (2020). <https://doi.org/10.1145/3373087.3375367>
12. Norouzi, A., Rahim, M., Altameem, A., Saba, T., Ehsani Rad, A., Rehman, A., Uddin, M.: Medical image segmentation methods, algorithms, and applications. *IETE Techn. Rev.* **31**, 199–213 (2014). <https://doi.org/10.1080/02564602.2014.906861>
13. NVIDIA Developer: Deep learning frameworks. <https://developer.nvidia.com/deep-learning-frameworks>
14. Shawahna, A., Sait, S.M., El-Maleh, A.: FPGA-based accelerators of deep learning networks for learning and classification: a review. *IEEE Access* **7**, 7823–7859 (2018)
15. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). <https://doi.org/10.48550/ARXIV.1409.1556>
16. Storn, R., Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces (1997)
17. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision (2015). <https://doi.org/10.48550/ARXIV.1512.00567>
18. Wang, J., Cong, J.: Automated generation of high-performance large-scale matrix multiplication accelerator on fpga (2016)
19. Yamuna Prasad, Kanad K. Biswas, Ā.K.J.: SVM classifier based feature selection using GA, ACO and PSO for siRNA design (2010)
20. Zeng, S., et al.: Enabling efficient and flexible FPGA virtualization for deep learning in the cloud (2020). <https://doi.org/10.48550/ARXIV.2003.12101>