Artificial Neural Network and Accelerator Co-design using Evolutionary Algorithms

1st Philip Colangelo
Intel PSG
San Jose, USA
philip.colangelo@intel.com

2nd Oren Segal
Hofstra University
Hempstead, USA
oren.segal@hofstra.edu

3rd Alex Speicher
Hofstra University
Hempstead, USA
aspeicher1@pride.hofstra.edu

4th Martin Margala
University of Massachusetts Lowell
Lowell, USA
Martin_Margala@uml.edu

***Abstract*—Multilayer feed-forward Artificial Neural Networks (ANNs) are universal function approximators capable of modeling measurable functions to any desired degree of accuracy. In practice, designing practical, efficient neural network architectures requires significant effort and expertise. Further, designing efficient neural network architectures that fit optimally on hardware for the benefit of acceleration adds yet another degree of complexity. In this paper, we use Evolutionary Cell Aided Design (ECAD), a framework capable of searching the design spaces for ANN structures and reconfigurable hardware to find solutions based on a set of constraints and fitness functions. Providing a modular and scalable 2D systolic array based machine learning accelerator design built for an Arria 10 GX 1150 FPGA device using OpenCL enables results to be tested and deployed in real hardware. Along with the hardware, a software model of the architecture was developed to speed up the evolutionary process. We present results from the ECAD framework showing the effect various optimizations including accuracy, images per second, effective giga-operations per second, and latency have on both ANN and hardware configurations. Through this work we show that unique solutions can exist for each optimization resulting in the best performance. This work lays the foundation for finding machine learning based solutions for a wide range of applications having different system constraints.**

***Index Terms*—evolutionary algorithm, machine learning, FPGA**

I. Introduction

The difficulty in designing performant NNAs has brought a recent surge in interest in auto design of NNAs. The focus of the existing body of research has been on optimizing NNA design for accuracy [1][2][3]. Optimizing NNAs is typically a difficult process in part because of the vast number of hyperparameter combinations that exist, and in cases where a combination is not optimal, performance will suffer. In fact, many deep learning frameworks such as TensorFlow [4] and Keras [5] offer support for hyperparameter tuning, but these are typically Bayesian optimizations that treat the ANN as a black-box and are used for the neural network training process. Research shows that the parameters of a network can directly influence the accuracy, throughput, and energy consumption of that model in deployment [6].

Once an accurate NNA has been found, the next step is to try to fit it into existing hardware i.e. a CPU, GPU, or a custom built but general purpose neural network hardware device such as a TPU [7]. None of these hardware solutions offer network specific specialization. The gap between the two optimizations is where ECAD comes in, it allows to search for an optimal hardware/NNA co-design by exploring the design space on the NNA and the hardware side and allows to implement a custom hardware solution for a specific NNA model using reconfigurable hardware.

In this paper, we provide the following contributions:

1) ECAD, a framework using evolutionary algorithms for the co-design of artificial neural networks and reconfigurable hardware design.
2) Examples and results optimizing for various fitness objectives including top accuracy, images per second, latency, and effective giga operations per second.

II. Related Work

In the past several years we have seen great strides made in the performance of ML algorithms on complex tasks using deep neural networks [8][9]. Manually designing state of the art deep neural network architectures for ML requires significant amount of time and labor [2][10]. Automating NNA search has been an ongoing effort for the past few decades but is becoming a focus of the NNA research community because of the difficulty in designing deep networks which are ever growing in complexity[2][3][10]. Automatic Artificial Neural Network Architectures Search (NAS) can be conducted using different strategies such as random search, evolutionary algorithms, Reinforcement Learning (RL), Bayesian optimization, and gradient-based methods [10]. Using Evolutionary Algorithms (EAs) to search for performant architectures has been investigated extensively [11][12] over the years. Some recent results indicate that evolutionary algorithms offer better results than random search and reinforcement learning [3] Recently, there has been growing interest in NAS for deep neural networks that specialize in image recognition [1][2][13].

As deep and complex neural networks became increasingly popular and with the realization that existing hardware architectures are not specifically optimized for such computation, new forms of specialized architectures have been proposed and designed [7] to help increase performance and energy efficiency. Designing such static new architectures can be prohibitively expensive and risky since the field of neural computing is evolving so rapidly. Optimizing hardware for neural networks is a research topic that is constantly evolving

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

and correlating with the ever-changing network structures that are being developed. Recent publications have shown new architectures carving their niche in deep learning by offering unique methods for accelerating the workloads of neural network applications [14][15][16]. Specifically, these reconfigurable architectures are a popular platform for both research and deployment due to their ability to change their fabric routing and resource structures to fit various workloads and optimizations by leveraging the resiliency of neural networks through low-numeric precision [17][18] and sparsity [19][20]. Other accelerator designs use reconfigurable fabrics to change the logic routing to preprocess data and offload to a specialized ASIC [21].

Multiple tool flows exist for optimizing fixed NNA designs for reconfigurable hardware (FPGAs). The majority of available tool flows target image recognition tasks. A recent survey on available tool flows is available here [22].

The body of work on NAS concentrate on accuracy as the main measure of performance, though optimizing for NAS can lead to more simplified NNA that could in turn simplify and optimize hardware designs [3][10]. On the other hand, optimizing for hardware performance parameters (latency/throughput/power) is normally done on an existing NNA design and there is no attempt to modify the NNA (layers/neurons etc.)[22].

Combining NAS and hardware optimizations could potentially close the loop between design and implementation of NNAs[22].

To the best of our knowledge ECAD is the first framework capable of conducting NAS and hardware co-optimization. It is capable of working on both the NNA level (neurons/layers etc.) and the hardware level (LUTS/DSPs etc.) at the same time i.e. given a general NNA structure it will evolve and search both spaces (NNA/hardware) in tandem.

III. ECAD Software

ECAD is intended to create a NNA that is optimized towards specific design goals. At the heart of the software side, lies an evolutionary algorithm and a vector of fitness functions. Fitness functions currently include measurements of accuracy, speed, energy efficiency, and throughput. ECAD allows to select the importance (weight) given to each of the fitness functions and by doing so guide an evolutionary process towards the required fitness goals. The result is a neural network design optimized towards the goals specified in the fitness functions. Figure 1 shows an overview of the ECAD flow. The next sections provide detail for each of the stages in the flow.

Population Generation Initially the system will create a population of neural networks using the base design specified in a configuration file. The population initial size, maximum size and change rate are all controlled using the configuration parameters. Each auto generated network instance will be mutated and different from the original base design.

As the evolutionary process progresses and once a sufficient number of networks are evaluated according to all fitness parameters, the process of population generation will repeat itself continuously except that the mutations will be based on the most fit individuals in the population, selected according to their fitness scores (see steady-state model in [23]).

Testing Population Fitness In the next stage each NN instance will be sent to one or more fitness evaluators or workers in ECAD terminology. The workers are designed as independent processes and can be distributed across a cluster of computers. They are orchestrated using a Master/Worker parallel computation model [24] running on top of MPI [25].

Each Worker sits in a separate process, inherits from a common C++ Worker class, and implements a common software interface. The system is built to be flexible and allow adding different types of workers easily. The implementation details for each worker can be completely different.

We currently have three types of workers implemented:

1) *Simulation Worker* capable of simulating a NN design and return accuracy and timing results
2) *Physical Worker* capable of synthesizing a NN design and return hardware synthesis results
3) *HWDB Worker* capable of accurately estimating NN synthesis results and return estimated hardware synthesis results

Once a worker's fitness evaluation is complete it will return the results to the master/server process. The master process will collect the results from all the workers that evaluated each NN and apply a combined score to each NN. Scores are combined using a unique Id that is assigned to each generated NN instance when it is initially created.

Sorting Population by Fitness ECAD will sort the NN population according to its score and the top instances will be mutated. Mutated NNs will be introduced into the population and sent to be evaluated, this process will continue until the end condition is met. End conditions can be the number of generations the simulation flow is requested to run or a desired fitness is met.

Neural Network Simulator The Neural Network Simulator (NeuralNetSim) is responsible for testing and verifying the ECAD neural network architectures. We chose TensorFlow[4] as the machine learning simulation framework. The NeuralNetSim consists of an ECAD Reader which servers as an interface to the evolutionary framework. Its main responsibility is accepting inputs such as files and training arguments as well as returning data and reports after training has been completed. It extracts the network info from the ECAD file and passes it to the TF Model Builder. TF Model Builder holds the actual TensorFlow graph and all other TensorFlow variables. It is responsible for dynamically creating the graph based on the info passed by the reader. Finally, it also holds the training functions which are called from the ECAD reader. TF Functions are a collection of TensorFlow API functions that are called by the TF Model Builder when building the graph.

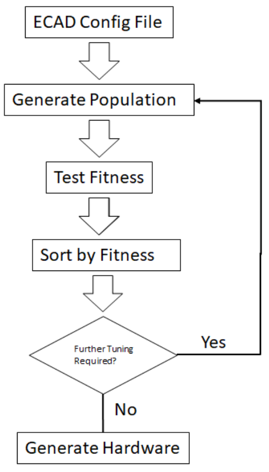

Fig. 1. ECAD flow

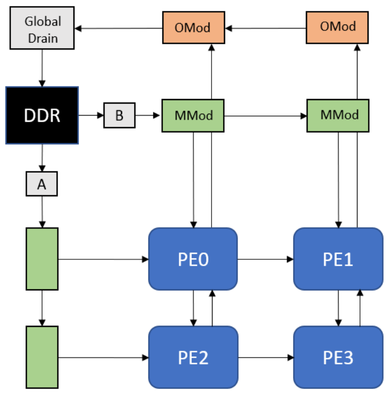

Fig. 2. 2D systolic array hardware architecture used in the design space exploration

IV. ECAD Hardware: 2D Systolic Array Implementation

Systolic arrays (SA) [26] are a great fit for FPGA because of their pipelined data flow and memory bandwidth tuning capabilities. The high level design shown in Figure 2 provides an overview of the architecture used by ECADs HWDB Worker. The following sections describe the architecture in detail.

A. Matrix Blocking and Memory Considerations

Matrices are blocked against spatial configurations of the SA as shown in Figure 3 and read via loader kernels depicted as A and B in Figure 2. Matrix A block height is defined by the number of rows in the SA multiplied by an interleaving factor. The height of a matrix B block is the same as the width of a matrix A block and is defined by the product of vector width and scaling factor. The width of matrix B is defined as the number of columns in the SA multiplied by an interleaving factor. Interleaving is a parameter that is adjusted for balancing data reuse to ease bandwidth constraints. The larger the interleaving factor, the more data reuse and less bandwidth that is required to feed the PEs, but as the block size grows it becomes less efficient for mapping to smaller matrix sizes. Scaling factor is a parameter that is used to enable more efficient global memory access. We treat global memory as a 512-bit cache line. Every trip to global memory reads in a vector width amount of data and if the vector width is less than 512-bits then this can lead to sub-optimal bandwidth utilization. Scale will tune the block width so that each read to global memory is closer to 512-bits and more efficient. Blocks are stored contiguously in DDR memory to ease the global memory access patterns. Because of this, we transpose matrix B on the host so that sequential memory accesses traverse its rows.

B. Memory Modules

Memory modules (MMods) are nothing more than a daisy chain of smart double buffers that read in the next block of data from the loader modules into a local cache while writing the current block to the PEs. MMods are chained along both the row and column dimension with the outer most module connected to a loader. Following Figure 4, an MMod is made up of an input router whose job is to first direct a block of input to the write select demux before switching over to sending data down to its neighbor MMod. Once a cache such as Mem0 is full, the buff_sel select will update so that new blocks are written to one memory while the read mux takes data from the other. Both memories are arbitrated in such a way that no read or write contention exists. All non-select lines depicted in Figure 4 have a width that supports a vector worth of data.

C. Processing Elements

Each PE is responsible for computing a dot-product. Peripheral PEs (PE0, PE1, PE2 as seen in Figure 2) get one input from a memory module (MMod) and the other from a neighbor except for the very first PE (PE0) who receives both inputs from MMods. Inner PEs (PE3) receive their input from both neighbor PEs. Each connection to a PE (except for the OMod connections that will be covered shortly) carries several data elements equal to the vectorization parameter of the array, or in other words, the width of the dot product. Figure 5 shows the internal workings of a PE. Each multiplier and adder in our design computes on 32-bit single-precision floating point data. The width of the dot product is depicted as n in the diagram. We chose a reduction tree strategy to make effective use of the DSP blocks and allow for deep pipelining of the design. PEs also include a small cache shown as shift registers (SR) in Figure 5. The size of the shift registers is based on the interleaving factor (shown as I) Every cycle, a new vector enters the tree and is accumulated along with a previous value that is stored in one of the shift registers. The output from this accumulation is then routed to either the output or to the back of the shift register. The demux selector is based on a counter that keeps track of how many partial sums have been computed which signals when a result is ready. Once the counter rolls

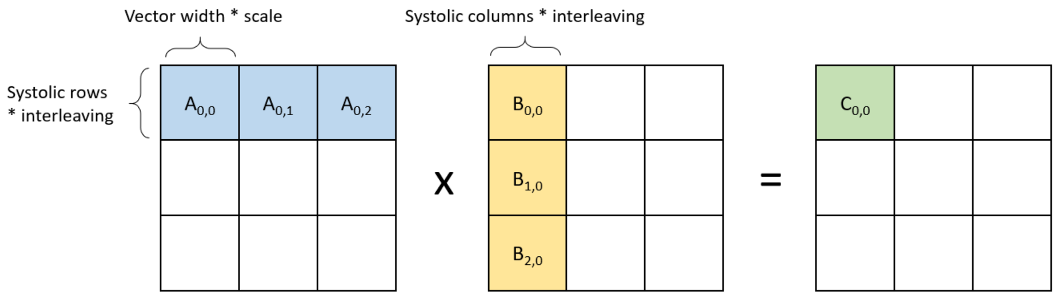

Fig. 3. Matrix blocking example.

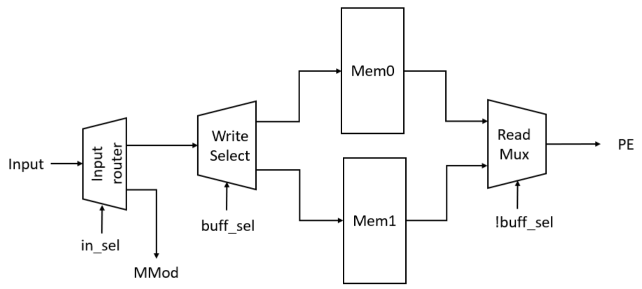

Fig. 4. Memory module internals.

over, a drain sequence begins by routing the accumulated result out and starting a new output block sequence.

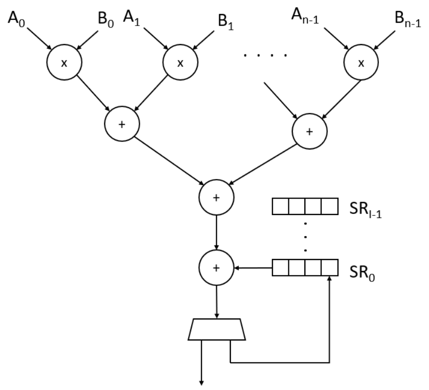

Fig. 5. PE internals.

D. Output Modules and Global Drain

Once a block of data is ready to be saved back to global memory, the PEs start the draining process which begins by writing the contents of its cache to its neighbor. Results are drained in rows, so each PE drains along its column. The first row of PEs is connected to an output module (OMod) which are connected to each other in a daisy chain fashion (refer to Figure 2). OMods continue the draining process by propagating the results along to a global drain whose responsibility is to prepare the data to be written back to global memory. Data being drained arrives to the output modules in a non-contiguous way, so the global drain has its own local cache that is used to buffer data back to DDR, see Figure 6. While this is the base design used for all our experiments, the global drain does require additional memory resources, so when scaling designs, the reordering of data may need to be done back on the host processor. Reordering data via global memory addressing is not efficient so we always write data back in a contiguous fashion. The global drain supports some additional features unique to traditional MLP style of compute. Both bias and activation function support is included and optionally bypassed if desired. In the case of bias skipping, we preload the bias cache with all zeros and never read from global memory. When bias is used, enough bias data is prefetched into a small local cache to be used on the next drain sequence. For the activation function block, we bypass simply using the activation mux as shown in Figure 6.

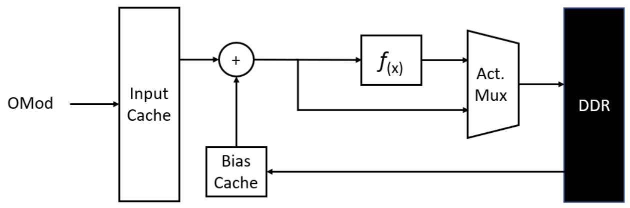

Fig. 6. Global drain internals.

E. OpenCL Implementation of the 2D Systolic Array

Each module described in the previous sections was coded in OpenCL as a separate kernel and connected using Intel's OpenCL channels extension. Hooks into the modular design were made available via the following C99 preprocessor macros: SYS_ROWS, SYS_COLS, SYS_VEC, INTERLEAVE, and SCALE. Each macro affects what hardware is generated, and so, various permutations of these macros can be combined to generate unique hardware.

F. Hardware Model

ECADs HardwareDB Worker (HWDB) is given a permutatioin of the SA described in the previous sections. Starting with an accelerator description that consists of a memory type along with its capabilities such as data rate and number of modules, and an FPGA device with target F_{max}, the EA will send a request to the HWDB consisting of an ANN description, and hardware configuration that returns projections for all supported metrics. Metrics that were returned during all our experiments include images per second (img/s), latency, and effective giga-operations (EGOP/s).

V. Evaluation

We present a series of experiments that show how both ANN and hardware configurations can be molded to fit different optimizations and that ultimately one solution cannot satisfy all design search spaces. First, we validated the hardware model by measuring hardware and comparing the total time it takes to classify MNIST images with the projections. We then use ECAD to search for optimizations by modifying various hardware parameters to gain an understanding about their effect on the search. There are four optimizations that we considered: accuracy, latency, img/s, and effective giga-operations per second (EGOP/s). The accelerator we used for all hardware results was an Arria 10 GX 1150 FPGA development kit by Intel. This is a mid-range device with 1 bank of DDR4 memory offering a peak bandwidth of 19.2 GB/s and peak performance of 1.5 TFLOP/s [27]. During search, modeled results are based off an F_{max} of 250 MHz but for measured comparisons, results are normalized to the hardware F_{max}.

A. Hardware Only Search

Without pressure on accuracy, the EA is completely biased towards hardware and will only use the ANN parameters to help tune the accelerator results. The motivation for running such experiments is to gain insights into how an ANN structure might look under certain performance optimizations and to understand how hardware configurations converge. Despite searching without accuracy, there is still a constraint on the maximum number of neurons that can be used per layer. All results in this section can be found in Table I. We searched over a 3-layer multi-perceptron network with the two hidden layers described by their number of neurons respectively as shown in the table. In some cases, as listed in the optimization for img/s using 1 DDR bank, a result can be bandwidth limited meaning that performance degrades due to the ratio between the available and required data per cycle. As we will see in the next section, despite a degradation due to bandwidth limitations, some configurations will still succeed in the evolution process due to the fitness success for the optimization. Our performance modeling was still found to be accurate for these cases.

Images per second The goal for img/s is to find an accelerator configuration that processes a single batch of images the fastest and it will follow that this configuration will process the most images in one second. This optimization is most useful for when the highest performing hardware for an ANN is desired. Looking at the data, the hardware configurations for img/s are nearly identical yet better performance is listed for the 2 DDR bank solution. Bandwidth comes into play here as the hardware configuration for img/s for a single bank was bandwidth bound. In this case, by supplying more memory bandwidth to the accelerator, the pipeline was alleviated from potential stalls and so better performance was achieved. Typically, as evident by the reported data, configurations with more available bandwidth tend to perform better because more extreme configurations won't be stalled as we will see in the next section on latency.

Latency During the evolutionary process, latency is returned as the amount of time in milliseconds it takes to process a single image. Of course, the goal being to minimize this number. Much like img/s, the best latency was achieved with more available bandwidth. With each permutation, the EA aimed to reduce the size of the hardware by decreasing the configuration parameters. Inherent in the SA design are certain parameters that help balance bandwidth and computation and once those parameters are lowered to a point, the computational demands become too much for the bandwidth to handle without stalling. Most notably, the interleaving factor was able to be reduced to 2 for the higher bandwidth configuration while the lower bandwidth configuration settled on an interleaving factor of 4. Interleaving plays a role in how many cycles a piece of data is used for and so the larger the interleaving, the more computational cycles the processing elements stay busy before needing new data, however the trade-off being that this also drives the latency up.

Effective GOP/s Efficient use of an accelerator is always desired because any unused, available logic can be considered wasted performance or cost. We include EGOP/s in our searches to allow pressure to find designs that better utilize the accelerator logic. Every design has a potential giga-operations based directly on the hardware configuration and EGOP/s based on the problem size being mapped onto the accelerator. While img/s aims to find the best performance overall, EGOP/s will balance the performance with efficient usage of logic area. This type of search can be useful if power is a concern or if the hardware can only allocate so much space for the NNA acceleration. What sets these results apart can be mainly seen in the neurons which unlike img/s and latency, seems more realistic if accuracy were a concern. However, as we will see with pressure on accuracy, this effect may not produce the most optimal trade-off between accuracy and number of neurons. The hardware configurations that converged for EGOP/s are also a lot larger in columns, interleaving, and scale leading to higher latency which may allude to an inverse relationship between the two optimizations.

B. Hardware Search with Simulator Accuracy

Bringing the Simulator back into the evolutionary process adds weight to the ANN accuracy and so the results become a combination between ANN and hardware performances. We searched over a 2-layer multi-perceptron network optimizing for both accuracy and img/s. Keeping track of each generation allows the evolutionary process to be tracked, sorted and

TABLE I
EA Results for Hardware Only Optimizations using Single and Dual Bank DDR Memory

Optimization	# Banks	Batch Size	Neurons	BW bound	Latency (ms)	Img/s	EGOP/s	Rows	Cols	Vec	Interleave	Scale
Img/s	1	992	48; 180	Yes	0.068352	1,243,182.00	119	4	8	16	8	2
Latency	1	2	18; 142	No	0.01792	111,607.00	4.0375	2	4	8	4	2
EGOP/s	1	958	992; 1010	No	3.6495	47,508.98	170	4	16	8	16	30
Img/s	2	992	62; 190	No	0.06348	1,532,832.00	191	4	8	16	8	4
Latency	2	158	2; 64	No	0.008096	505,425.00	2.36	2	2	4	2	2
EGOP/s	2	960	998; 1014	No	2.765	49,792.86	179	4	16	8	16	32

TABLE II
Top permutations from optimizing accuracy and img/s

Top Permutation	Fitness Sum	Accuracy	Batch size	Neurons	HW Config.	Latency (ms)	Img/s	EGOP/s
Fitness Sum	1.065	0.936	508	852	2,8,32,16,2	0.352	129,056	174.61
Accuracy	1.006	0.942	484	1018	2,8,16,16,2	0.583	57,296	92.62
Latency (ms)	1.065	0.936	508	852	2,8,32,16,2	0.352	129,056	174.61
Img/s	1.064	0.935	572	970	2,16,32,32,2	1.283	129,144	198.93
EGOP/s	1.062	0.933	572	980	2,16,32,32,2	1.283	129,144	200.98

evaluated. Results for the top permutations found during this search are presented in Table II. We add top permutations for other metrics (despite not explicitly optimizing) to provide a contrast to the top performer. Row 1 shows the top performing permutation based off the fitness sum which considers the weighted accumulation of accuracy and img/s. Contrasting this result with the top permutation for accuracy we can see a slight trade-off of 0.6% accuracy for an img/s speed up of ~2.25x. In the previous section we showed results for latency and how the EA reduced both the ANN and the hardware configurations, however, with pressure on maintaining accuracy, this can only be done to a limited extent and it happens to show that the top performer for latency is the same as the overall top performer. Had latency been the target optimization then this probably would not have the same result. We hypothesize that results obtained with pressure on accuracy and with good latency will be very efficient since the goal is to find the smallest and most accurate design. Similarly, comparing results for img/s and EGOP/s resulted in almost the same design, the difference being in the ANN. EGOP/s has a lower fitness sum due to desiring more neurons to obtain a better fit to the hardware, but as it turns out, was necessary due to the ANN training procedure finding a better accuracy with 10 less neurons.

As part of this experiment, we compiled a couple of the top performers to hardware and measured the results to valid the hardware model. Table III contains the measured results and Table IV contains the modeled results for comparison. The measured time included the total execution time to run a single batch (size found in Table II) of MNIST images. Not listed in the tables but verified is the accuracy, i.e., our accelerator imposed no degradation to accuracy. Additionally, provided as a percentage is the resource breakdown for ALM, SRAM, and DSP usage. During the evolutionary process, the HWDB (optionally) does not consider ALM, but it does attempt to weed out designs that cannot fit the necessary SRAM caches and is able to determine the exact number of DSP blocks a design requires. As part of a post-processing and data analysis step, we go through top performers to determine which results are synthesizable by first doing partial compiles, these results are the resource utilizations found in Table IV. The top performers all had similar configurations, so we handpicked the 4,8,8,16,18 configuration as an additional validation for the hardware model. F_{max} in Table IV is there as a reminder that F_{max} was normalized to hardware for the comparison.

TABLE III
Measured hardware performance results

HW Config.	Time (ms)	ALM	SRAM	DSP	F_{max} (MHz)
2,8,16,16,2	9.64	37%	36%	26%	220
2,8,32,16,2	5.53	50%	51%	43%	212
4,8,8,16,18	4.65	38%	34%	26%	228

TABLE IV
Modeled hardware performance results

HW Config.	Time (ms)	ALM	SRAM	DSP	F_{max} (MHz)
2,8,16,16,2	9.59	47%	38%	26%	220
2,8,32,16,2	4.64	61%	55%	43%	212
4,8,8,16,18	4.66	45%	37%	26%	228

VI. Conclusions

We present a novel framework (ECAD) for the exploration and design of Neural Network Architectures(NNAs) based on evolutionary algorithms. We use a reconfigurable-hardware/software co-design approach to optimize NNA designs on both the software and the hardware side. We discuss and demonstrate the possibility to optimize NNA designs for different and multiple optimization objectives instead of the predominant approach to focus on accuracy optimization alone. Performance data of the top permutations for different optimizations including accuracy, img/s, and both simultaneously, is presented along with the complete end-to-end ECAD flow targeting an Intel FPGA Arria 10 1150 GX device. Finally, comparing the hardware performance of the top permutations against our model served to validate the results of the evolutionary process.

References

[1] Hanxiao Liu et al. "Hierarchical representations for efficient architecture search". In: *arXiv preprint arXiv:1711.00436* (2017).

[2] Esteban Real et al. "Large-scale evolution of image classifiers". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2902–2911.

[3] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *arXiv preprint arXiv:1802.01548* (2018).

[4] Martin Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th Symposium on Operating Systems Design and Implementation 16*. 2016, pp. 265–283.

[5] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[6] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. "An analysis of deep neural network models for practical applications". In: *arXiv preprint arXiv:1605.07678* (2016).

[7] Norman Jouppi et al. "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3 (2018), pp. 10–19.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[9] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[10] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey". In: *arXiv preprint arXiv:1808.05377* (2018).

[11] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. "Designing Neural Networks using Genetic Algorithms." In: *ICGA*. Vol. 89. 1989, pp. 379–384.

[12] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[13] Barret Zoph et al. "Learning Transferable Architectures for Scalable Image Recognition". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 8697–8710.

[14] Utku Aydonat et al. "An OpenCL(TM) Deep Learning Accelerator on Arria 10". In: *CoRR* abs/1701.03534 (2017). arXiv: 1701.03534. URL: http://arxiv.org/abs/1701.03534.

[15] Naveen Suda et al. "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: ACM, 2016, pp. 16–25. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847276. URL: http://doi.acm.org/10.1145/2847263.2847276.

[16] Jinhwan Park and Wonyong Sung. "FPGA based implementation of deep neural networks using on-chip memory only". In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*. 2016, pp. 1011–1015. DOI: 10.1109/ICASSP.2016.7471828. URL: https://doi.org/10.1109/ICASSP.2016.7471828.

[17] Philip Colangelo et al. "Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs". In: *CoRR* abs/1806.11547 (2018). arXiv: 1806.11547. URL: http://arxiv.org/abs/1806.11547.

[18] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *CoRR* abs/1612.07119 (2016). arXiv: 1612.07119. URL: http://arxiv.org/abs/1612.07119.

[19] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. "Accelerating Deep Convolutional Networks using low-precision and sparsity". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*. 2017, pp. 2861–2865. DOI: 10.1109/ICASSP.2017.7952679. URL: https://doi.org/10.1109/ICASSP.2017.7952679.

[20] J. Yinger et al. "Customizable FPGA OpenCL matrix multiply design template for deep neural networks". In: *2017 International Conference on Field Programmable Technology (ICFPT)*. Dec. 2017, pp. 259–262. DOI: 10.1109/FPT.2017.8280155.

[21] Eriko Nurvitadhi et al. "In-Package Domain-Specific ASICs for Intel®Stratix®10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC(Abstract Only)". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. Monterey, CALIFORNIA, USA: ACM, 2018, pp. 287–287. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174966. URL: http://doi.acm.org/10.1145/3174243.3174966.

[22] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 56.

[23] David E Goldberg and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algo-

rithms”. In: *Foundations of genetic algorithms*. Vol. 1. Elsevier, 1991, pp. 69–93.

[24] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013.

[25] Edgar Gabriel et al. “Open MPI: Goals, concept, and design of a next generation MPI implementation”. In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2004, pp. 97–104.

[26] HT Kung and Charles E Leiserson. “Systolic arrays (for VLSI)”. In: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for industrial and applied mathematics. 1979, pp. 256–282.

[27] *Intel Arria10 Device Overview*. 2018. URL: https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10/features.html.