EvoSh: Evolutionary Search with Shaving to Enable Power-Latency Tradeoff in Deep Learning Computing on Embedded Systems

Basar Kutukcu*, Sabur Baidya†, Anand Raghunathan‡, Sujit Dey*
*Department of Electrical and Computer Engineering, University of California, San Diego, CA, USA
†Department of Computer Science and Engineering, University of Louisville, KY, USA
‡Department of Electrical and Computer Engineering, Purdue University, IN, USA
e-mail: bktkc@ucsd.edu, sabur.baidya@louisville.edu, raghunathan@purdue.edu, dey@ucsd.edu

***Abstract*—Deploying deep-learning applications on resource-constrained embedded systems requires exploration of large design spaces for mapping (to heterogeneous processing units) and hardware configuration selection (voltage/frequency levels) to balance power consumption and latency. Herein, we propose a search algorithm called Evolution with shaving to find optimized configurations in these search spaces. We evaluate our approach using 3 state-of-the-art image classification DNNs on Nvidia Jetson-TX2 platform, demonstrating the benefit of exploring the proposed search spaces and the ability of the proposed algorithm to successfully perform the search. We show that our approach achieves optimized mapping and hardware configuration in $<$0.1% of search-space exploration.**

***Index Terms*—Neural networks, embedded systems, design space exploration**

I. Introduction

Deep learning algorithms have achieved remarkable success in various machine learning tasks, driving great demand for implementing them on mobile and embedded systems [1]. However, implementing deep learning models on constrained embedded devices involves various challenges [2] in order to perform efficiently, especially, in terms of power consumption and latency of inference. The power consumption is an important factor for portable systems since they usually run on a battery with finite energy budget. The latency of the inference, on the other hand, is important for various mission-critical and real-time applications. As there can be different power and latency requirements depending on the system and the application being used, these two metrics usually create a tradeoff. Hence, it is important to find a configuration search space that provides a meaningful tradeoff to satisfy different requirements from the system and the application. There could be different mechanisms to achieve this tradeoff due to the modular architecture of deep learning algorithms, and the availability of heterogeneous computing units with different hardware configurations in embedded systems. In this paper, we propose to use two different search spaces and propose a systematic search method to explore the power consumption vs. latency tradeoff that they provide.

The first search space arises from dividing a deep learning application into blocks and mapping them to different computation units that exist in the system. These blocks can be as small as a layer of a neural network, or can be a combination of several consecutive layers. Selecting smaller blocks creates a finer precision in the power consumption – latency tradeoff, while increasing the size of the search space and therefore the time required for searching through it. The different computation units can include CPUs, GPUs, deep learning accelerators, or any other domain specific processing units. The range of parameters that can be varied in these computing units constitutes the second search space.

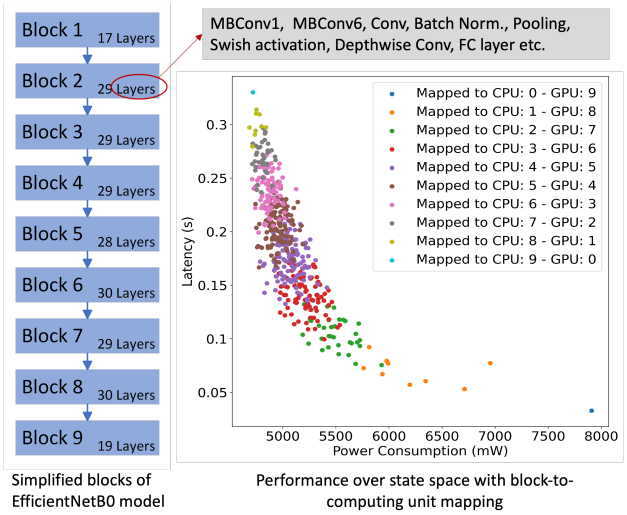


Fig. 1: The search space overview of 9-blocks EfficientNetB0

We illustrate the complexity of the search space using a small example where an exhaustive search is possible. We consider the *EfficientNetB0* image classification model as our deep learning application, and split it into 9 blocks as shown in Fig. 1. We are using CPU and GPU on an Nvidia Jetson TX2 as our computation units. This creates a search space with 512 ($= 2^9$) different mapping configurations. We ran each of these mappings permutations on the Jetson TX2 platform while measuring power consumption and inference latency, and plot the results in Fig. 1. In this plot, each color represents a fixed number of blocks that are mapped to the same computation unit. Each dot represents a unique mapping combination. For example, when 1 block is mapped to CPU and 8 blocks

are mapped to GPU, there are 9 possible different mapping configurations (represented by the 9 orange dots in Fig 1). Similarly, the mapping configuration with 2 blocks mapped to CPU and 7 blocks mapped to GPU has 36 possible different mappings (green dots in Fig 1). This plot clearly shows that different mapping configurations create a power consumption – latency tradeoff. Even though there is a general trend in the data, different mapping combinations are overlapped in the power consumption – latency space. More importantly, the specific blocks mapped to each computation unit matter, as there are good and bad performing configurations in the same color. So, finding the Pareto frontier (subset of solutions where each solution is optimal in terms of latency for some power threshold) is an important challenge in this problem.

Similar to the mapping configurations, the hardware configuration search space can be extremely large with respect to the number of parameters, e.g., the frequencies of computation units and memory controller, the number of CPU cores and types of CPU cores.

In order to efficiently search through the exponentially large search space mentioned above, several black box and white box algorithms are developed in the literature. Black box algorithms can be realized without any insight into the problem at hand, whereas white box algorithms may perform better due to their domain-specific techniques. Herein, we propose an algorithm that is based on a traditional evolutionary algorithm but enhanced with an intelligent heuristic that uses insights into the nature of the search spaces explored. We call this technique shaving, as it shaves some of the search space every step without trying them. We apply the proposed algorithm to the EfficientNet family of image classification models and the Nvidia Jetson TX2 platform. We showcase the advantage of our proposed approach to efficiently find the best mapping and hardware configurations for a given application requirement.

The main contributions of this work are as follows:

- We define two complementary search spaces - block-to-computation unit mapping, and hardware configuration selection - to provide a beneficial tradeoff between power consumption and inference latency for deep learning applications running on embedded systems.
- We present EvoSh - an intelligent configuration search algorithm that improves the traditional evolutionary search by using a novel shaving technique that leverages insights into the nature of the search space.
- The proposed configuration search spaces and the EvoSh algorithm are evaluated on the Nvidia Jetson TX2 platform, and the results demonstrate EvoSh outperforming other algorithms in wide range of experiments.

II. Related Work

Enabling deep learning inference on embedded devices has been tackled in the literature from different angles including quantization [3], pruning [4], adaptive models [5]–[7] or model selection [8], [9]. These methods improve inference efficiency by optimizing the model or using different models that perform the same task. Our work is complementary, which means we achieve a power consumption - latency tradeoff without changing the application structure. In other words, we do not trade off application accuracy for any gain.

Search algorithms have been instrumental to solutions of many problems in different fields in computer science, including computer systems and deep learning applications. Neural architecture search [10] has been an active research field where basic neural network blocks are defined, and a neural network architecture is searched using those blocks. The aim is to find a neural network architecture that achieves high accuracy on the given dataset with reasonable computational cost. This problem is different from ours as we do not change the neural network architecture, and focus on execution-performance related metrics.

There are other efforts focused on improving inference efficiency for a given model. In [11], hardware features of an in-house accelerator are searched to tradeoff latency and hardware area. They do not consider power, and they demonstrate their work on an in-house hardware which is proprietary. Moreover, they don't consider mapping of neural networks to multiple computation units. Similarly, the work in [12] searches through hardware features to design new hardware. Unlike previous works, they did not use simulation to evaluate their configurations. Instead, they used logged simulation data. They did not consider mapping as well, and hence, differ from our work. In [13], instead of neural networks, simpler operations, such as dot product or GEMM, are used. Various hardware and software features are searched on an FPGA platform, considering logic utilization and cycles. In [14], a general Pareto frontier identifier is proposed and applied to searching hardware and software compiler features. In [15], first execution times are estimated and then, layer to computation unit mapping is proposed using these estimates and a solver. This work does not consider hardware features of the platform. Also, they do not have the fine-grained mapping search space as ours, i.e. they limit the number of blocks to 3 in their experiments. Similarly, in [16], a deep learning model is cut into multiple sections and these sections are pipelined during execution. The cut points in the deep learning model are searched while considering speed and energy of the resulting pipelined model. Their search space is not fine-grained as ours and also they do not consider hardware configuration search space. In [17], the mapping of neural network layers onto multiple processors is examined. Even though they consider hardware features in their framework for profiling, they do not search optimized hardware features as we considered here.

There are some papers that focus on improving the training time of neural networks through searching various parameters. In [18], network architectures and hyperparameters for training are searched, considering accuracy of the network and the energy consumption on a custom accelerator. In [19], the network architectures, operating system and hardware configurations are searched considering accuracy and energy. These works differ from our work since we focus on inference of neural networks but not on training.

III. Optimization Problem for the Best Mapping and Hardware Configuration Search

In this paper, we intend to find a configuration that is fast (low latency) while satisfying the given power threshold. These two metrics - latency and power, are usually in direct conflict. More importantly, since there is a direct relationship between them, they are affected together by any configuration change. That means, if we increase the computation capability by changing a configuration, latency decreases and power consumption increases at the same time. However, if we increase computation capability more than our workload's requirement, latency and power consumption may remain the same. Therefore, we can say that this problem has a monotonic nature.

In both of our proposed search spaces, changing a configuration is affecting the computation power, as shown in Fig. 1. In the first search space of mapping blocks to computation units, when we map a block to GPU, computation power generally increases. This usually translates to smaller inference delay and larger power consumption. In the second search space of hardware configurations, we modify some hardware features. When we increase the frequency of any unit or number of cores of any CPU, it usually translates to larger computation capability, and therefore, smaller inference delay and larger power consumption. We use this monotonic behavior as a priori knowledge in our algorithm.

1) Optimized Mapping Configuration:

Here, we consider a problem where we find the optimized mapping of layers of N-block neural network to a GPU or CPU with a constraint on the power consumption, where $N \in \mathbb{N}$. We define binary variable x_i as

$$x_i = \begin{cases} 1, & \text{if } i_{th} \text{ layer is executed on GPU} \\ 0, & \text{otherwise} \end{cases}$$

Where $i \in [1, ..., N]$. Let $p : \{0,1\}^N \rightarrow \mathbb{R}_+$ denote the power consumption and $t : \{0,1\}^N \rightarrow \mathbb{R}_+$ denote the execution time of the corresponding mapping. We have a constraint in the form of $p(x) \leq \bar{P}$, where $\bar{P}$ denotes the power threshold. We aim to find the mapping of layers to the computation units with minimum execution time, and thus we are interested in solving the following optimization problem:

$$\min_{x \in \{0,1\}^N} t(x) \quad (1)$$
$$s.t.\ p(x) \leq \bar{P}$$

2) Optimized Hardware Configuration:

In the second search space, we consider a problem where we find the optimized combination of N different hardware configurations with a constraint on the power consumption, where $N \in \mathbb{N}$. We define integer variable x_i as

$$x_i \in [1, ..., m_i]$$

where m_i is the maximum possible value for x_i, and $i \in [1, ..., N]$. Let $p : \{x_i \in [1, ..., m_i]\}^N \rightarrow \mathbb{R}_+$ denote the power consumption and $t : \{x_i \in [1, ..., m_i]\}^N \rightarrow \mathbb{R}_+$ denote the execution time of the corresponding hardware configuration combination. We have a constraint in the form of $p(x) \leq \bar{P}$, where $\bar{P}$ denotes the power threshold. We aim to find the hardware configuration combinations with minimum execution time, and thus we are interested in solving the following optimization problem:

$$\min_{x_i \in [1, ..., m_i]^N} t(x) \quad (2)$$
$$s.t.\ p(x) \leq \bar{P}$$

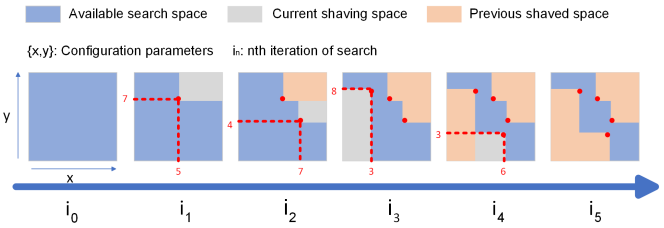


Fig. 2: Two dimensional shaving technique illustration

IV. Solution Methodology

1) Shaving Technique Insights:

We use the monotonic nature of our problem described earlier to accelerate the evolutionary search. At every step, the evolutionary search proposes a new configuration. After this proposition, we shave a part of the search space, i.e. mark that part as unsearchable for future steps. For this example, consider Fig. 2. Let us suppose we are searching the space of CPU and GPU frequencies and each of these has 10 different frequencies. The square blocks show the current state of the search space, x and y shows the CPU and GPU frequencies, and the overall figure shows the 4 iterations of shaving in Fig. 2. In our notation, 0 denotes the smallest frequency and 9 denotes the highest one. So, if the evolutionary search proposes the configuration (5,7) at iteration i_1, that means CPU and GPU are using the frequencies that are indexed at 5 and 7. So, if this configuration uses more power than the power threshold, we remove (6,7), (7,7), (5,8), (5,9) and so on. In other words, we remove every combination that is bigger than (5,7) that corresponds to the gray region at iteration i_1. We remove those since they should consume more power than (5,7) which already exceeds the power threshold. Note that, in the removed configurations, every parameter needs to be greater than or equal to their corresponding parameter in the proposed configuration. Similarly, consider iteration i_3 where the algorithm proposes (3,8). However, this time, let's assume this configuration has power consumption less than power threshold. Therefore, we remove permutations that are smaller than (3,8), such as (2,8), (1,8), (3,7), (3,6), (2,5) and so on. These permutations correspond to the gray area at iteration i_3. Those are removed since they should be slower than (3,8). The other steps have similar shavings in Fig. 2. As a result, we decrease the search space from blue region at iteration i_0 to blue region at iteration i_5 using just 4 steps. All the orange area is shaved from search space without being benchmarked on the board. The same algorithm works for mapping search space as well, where variables can be 0 (CPU) or 1 (GPU) for each layer that we are mapping.

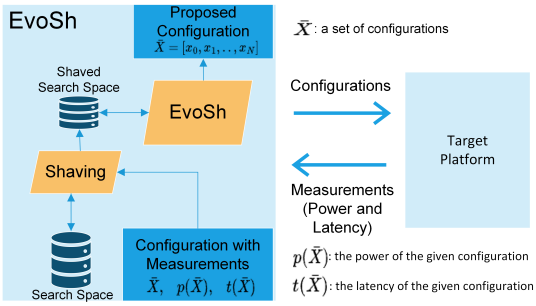


Fig. 3: Overview of EvoSh-based config-search framework

2) Evolutionary Algorithm with Shaving (EvoSh):

In Algorithm 1, we describe our general algorithm, EvoSh. Lines 1-8 explain the inputs and the data structures used in the algorithm. Lines 9-12 are for generating the initial population of the evolutionary algorithm. Lines 13-25 describe the iteration steps where the evolution and shaving occur. At line 14, a subpopulation is sampled from the main population to prevent the overfitting. At line 15, the best two configurations that satisfy the power threshold in subpopulation are found. At line 17, a new configuration is created from the two best configurations. Crossover rate defines the probability of picking a parameter from each parent configuration. The mutation rate defines the probability of changing the newly created parameter of the new configuration. Lines 17-20 describes the necessary code for creating a valid new configuration. At line 17, we check if the new configuration is already in the population or in the shaved part of the search space. As long as the condition at line 17 is true, we keep creating a new configuration while increasing the mutation rate. Once we find a valid new configuration, we add it to the population (line 21) and run it on the board to measure performance (line 22). Then, we shave the search space based on the new configuration results (line 23) and remove the oldest configuration from the population (line 24). We run this loop for a predefined number of steps. Increasing the number of steps increases the search time, but also leads to relatively better results.

3) EvoSh-based Optimized Config-search Framework:

Fig. 3 shows our general framework, including the algorithm side and measurement side. In the measurement side, we have our target board which receives the configurations from algorithm side, runs the configuration on the board while measuring power and speed, and sends the results back to the algorithm side. The algorithm side is running on the host computer. It receives the results from measurement side, shaves the search space, and runs the EvoSh, and proposes a new configuration to measurement side. This loop is run for a predefined number of steps.

V. Experiments

We use Nvidia Jetson TX2 and TensorFlow to implement the deep learning models and heterogeneous mapping to the hardware. We use the EfficientNet [20] family of image classification models for the experiments. However, our methodology is not tied to EfficientNet architecture and can be used for

Algorithm 1: EvoSh

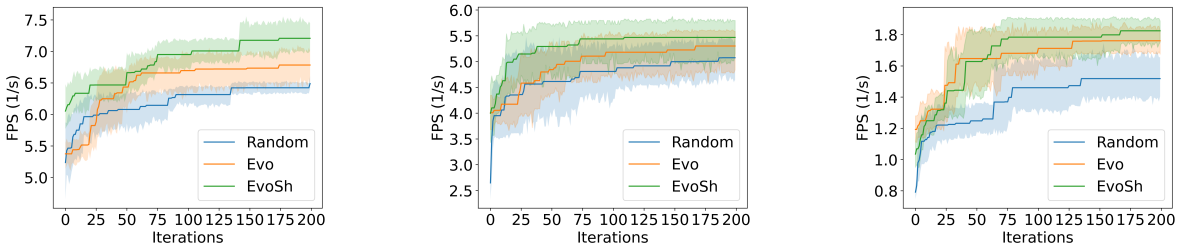
1: $pop\_size \leftarrow$ population size
2: $n \leftarrow$ number of parameters
3: $pop \leftarrow$ population – a list to hold pop_size of configurations
4: $perfs \leftarrow$ a dictionary, keys are configurations, values are tuples of power and latency
5: $num\_steps \leftarrow$ number of steps to run the algorithm
6: $pw\_th \leftarrow$ power threshold to satisfy
7: $co\_rate, mu\_rate \leftarrow$ crossover, mutation rates for evolution algorithm
8: $shaved \leftarrow$ a list to hold shaved configurations from search space
9: **for** $iteration = 1, 2, \ldots, pop\_size$ **do**
10: $\quad configuration = random\_vector(size = n)$
11: $\quad perfs[configuration] = run\_on\_board(configuration)$
12: **end for**
13: **for** $iteration = 1, 2, \ldots, num\_steps$ **do**
14: $\quad sub\_pop = sample(pop, sub\_pop\_size)$
15: $\quad w1, w2 = find\_best\_two(sub\_pop, perfs, pw\_th)$ ▷ returns the fastest two configurations that satisfy the power threshold
16: $\quad c = create\_child(w1, w2, co\_rate, mu\_rate)$
17: $\quad$ **while** c in pop or c in $shaved$ **do**
18: $\quad\quad c = create\_child(w1, w2, co\_rate, mu\_rate)$
19: $\quad\quad mu\_rate += 0.05$
20: $\quad$ **end while**
21: $\quad pop.add(c)$
22: $\quad perfs[c] = run\_on\_board(c)$
23: $\quad shave(c, perfs, pw\_th, shaved)$
24: $\quad pop.remove\_oldest()$
25: **end for**
26: **procedure** SHAVE($indiv, perfs, pw\_th, shaved$) ▷ indiv is taken as a pivot and all possible combinations either above or below of it are shaved depending on power consumption of it.
27: $\quad maxes \leftarrow$ a vector (elements are maximum possible parameters)
28: $\quad mins \leftarrow$ a vector (elements are minimum possible parameters)
29: $\quad pw\_indiv = perfs[indiv][0]$
30: $\quad$ **if** $pw\_indiv \geq pw\_th$ **then**
31: $\quad\quad starts, ends = indiv, maxes$
32: $\quad$ **else**
33: $\quad\quad starts, ends = mins, indiv$
34: $\quad$ **end if**
35: $\quad to\_remove = find\_combinations(starts, ends)$
36: $\quad shaved.add(to\_remove)$
37: **end procedure**

any deep learning model. In addition to proposed EvoSh, we evaluate random search as a baseline to demonstrate that our algorithm shows intelligence in its search. We also compare with conventional evolutionary search to show the benefits of shaving.

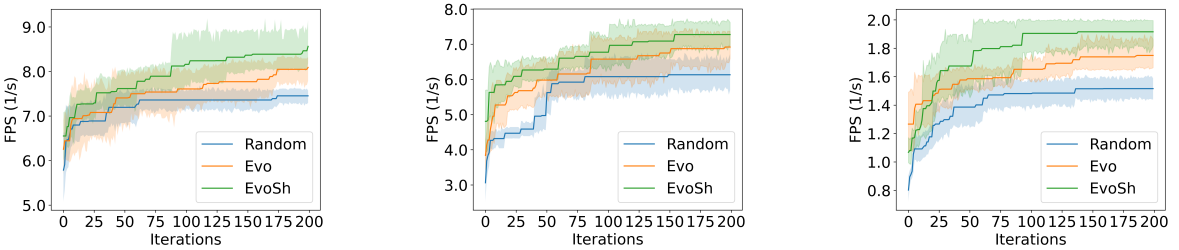
The real-time search is an expensive process in terms of time. Although, the search algorithms do not incur much time, the benchmarking of a proposed mapping or hardware configuration on the system requires much longer time considering it needs to be done for every iteration during the search. The reason is that we need to reload the required libraries and the model at every iteration to make the measuring process fair and consistent between runs. Then, we run the model for some warmup iterations and then start actual benchmarking, which needs to be long enough to measure the power accurately. This fact makes Shaving a valuable technique, as it removes many of the mappings or hardware configurations without actually running them on the target board. It is also important to note that our methodology is used during design time. Our method's aim is to find the optimized configuration that is going to be used in runtime.

A. Mapping Search Space

Nvidia Jetson TX2 has two computation units that are visible to TensorFlow - CPU and GPU. We can map TensorFlow layers to these computation units. In this section



(a) EfficientNetB0 - 5000mW (b) EfficientNetB2 - 6000mW (c) EfficientNetB4 - 6800mW



(d) EfficientNetB0 - 5200mW (e) EfficientNetB2 - 6500mW (f) EfficientNetB4 - 7000mW

Fig. 4: Performance of search algorithms for different deep learning models with different power thresholds

of experiments, we used EfficientNetB0, EfficientNetB2 and EfficientNetB4 as our applications. These three networks have similar architectures, but their architecture sizes and input sizes are different. EfficientNetB4 is the largest and using 380x380 images as input, EfficientNetB2 is the medium and using 260x260 images as input and EfficientNetB0 is the smallest and using 224x224 images as input.

We divide EfficientNetB0, EfficientNetB2, and EfficientNetB4 into 18, 20, and 34 blocks, respectively, where each of these blocks can be mapped to CPU or GPU. These create a search space of size of 2^{18}, 2^{20}, and 2^{34}, respectively. We show the performance of EvoSh and other algorithms on these search spaces in Fig. 4, using 2 power thresholds for each search space. Each of the algorithms is run 5 times to minimize the impact of random seeds. The solid line shows the mean of the 5 runs, and the shaded area shows the 95% confidence interval. Our algorithm consistently outperforms other ones, showing that shaving is a useful technique. Moreover, it shows improvement and convergence behavior in just 200 steps where there are 2^{18}, 2^{20}, 2^{34} permutations of mapping in total in the search spaces. This means we only explored 0.076% $(=(200/(2^{18})) * 100)$, 0.019% $(=(200/(2^{20})) * 100)$, 0.00000116% $(=(200/(2^{34})) * 100)$ and of the search space. This is because EvoSh removes many of the redundant configurations from the exploration.

Since our original mapping search spaces (18, 20, 34 blocks) are too large, it is impossible to run an exhaustive search to find the optimal values. Therefore, we give a simple motivational example to show convergence characteristics of our algorithm in Fig. 5. We have the exhaustively collected data for this search space because of the reasonable size (2^9 configurations). We set the power threshold to 5000mW and run each algorithm 20 times. The results are shown in Fig. 5 which shows that EvoSh can converge to the optimal configuration while the other algorithms converge to a suboptimal solution. Since the figure shows the mean of 20 runs, we can say that the results are consistent over multiple runs.

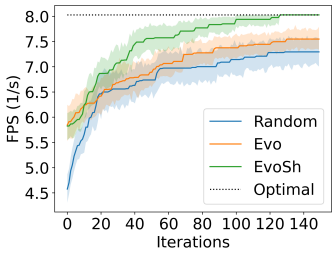
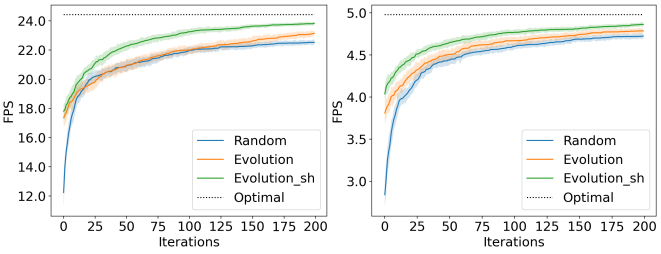
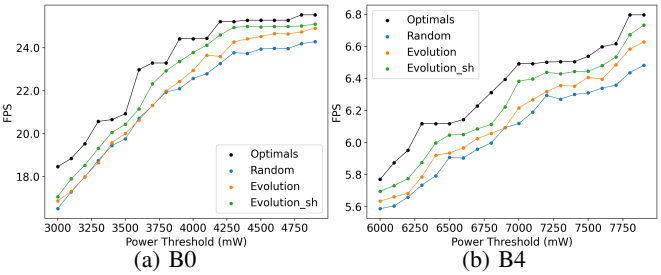


Fig. 5: Convergence characteristic of algorithms



(a) Power Threshold 4000 - B0 (b) Power Threshold 5000 - B4

Fig. 6: Performance comparisons of search algorithms on hardware configuration search space



(a) B0 (b) B4

Fig. 7: The power threshold sweep and the average performance of the algorithms compared to optimal results

B. Hardware Configurations

Nvidia Jetson TX2 allows users to modify some hardware configurations. The modifiable hardware features, the number and the range of possible values for each feature are given in Table I. We use EfficientNetB0 and EfficientNetB4 as our applications. We also did an exhaustive search for each of

Feature name	# of possible values (range)
# of A57 CPU cores	5 (0-4)
# of Denver CPU Cores	3 (0-2)
A57 CPU Frequency	12 (345MHz - 2GHz)
Denver CPU Frequency	12 (345MHz - 2GHz)
GPU Frequency	13 (114MHz - 1.3GHz)
EMC Frequency	11 (40MHz - 1.9GHz)

TABLE I: Nvidia Jetson TX2 Modifiable Hardware Features

the applications to show the performance of our algorithm. Note that exhaustive search takes a very long time and is not feasible to run for every network. The performances of EvoSh and the other algorithms can be seen in Fig. 6. The figures show the best FPS reached while satisfying power threshold until the corresponding iteration. Since we already had the data from the exhaustive search, we ran algorithms 100 times and plotted results with 95% confidence interval. The results show that EvoSh outperforms other algorithms. Moreover, we can reach a configuration that is very close to optimal in just 200 steps. Since we have exhaustively collected the data, we ran the algorithms many times by sweeping the power threshold for each of the neural networks. The results are shown in Fig. 7 which shows that EvoSh algorithm consistently outperforms the other algorithms for every power threshold and neural network combinations. Moreover, it can get very close to the optimal results for every power threshold in just 200 steps.

C. Discussion and Future Directions

In our experiments, EvoSh consistently outperforms the other algorithms and is shown to be capable of finding the optimal configuration where the search space is small enough for us to know which configuration is the optimal one. The improvement over other algorithms might seem marginal in terms of absolute values, i.e., 1-2 FPS, in the plots. However, these improvements are significant percentage wise(10%-30%).

The search space of our mapping problem is defined by the block sizes. The block size creates a tradeoff between granularity of the search space and the search time. Moreover, some splitting points for blocks might be invalid considering some architectural properties such as residual connections. Currently, our method takes the number of blocks and block size as input. In the future, we are planning to automate the block size definition process so that a good tradeoff point can be achieved without invalid split points.

VI. Conclusion

In this paper, we proposed two search spaces that provide beneficial tradeoffs between power consumption and inference latency of neural networks on embedded systems. Then, we proposed a novel search technique called EvoSh which uses the monotonic nature of our search spaces to improve upon evolutionary search. We evaluated our method on the Nvidia Jetson TX2 using a wide range of experiments. Our experiments show that our method consistently outperforms random and evolutionary search, which demonstrates the usefulness of the proposed shaving technique. EvoSh can quickly and successfully find configurations that satisfy various power thresholds. Our methodology can find near-optimal configurations by exploring a small fraction of the whole search space.

Acknowledgment

This work is partially supported by DARPA under grant number 304259-00001.

References

[1] T. S. Ajani, A. L. Imoize, and A. A. Atayero, "An overview of machine learning within embedded and mobile devices–optimizations and applications," *Sensors*, vol. 21, no. 13, 2021.

[2] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Comput. Surv.*, vol. 53, no. 4, 2020.

[3] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *CoRR*, vol. abs/2103.13630, 2021.

[4] D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag, "What is the state of neural network pruning?" in *Proceedings of Machine Learning and Systems, MLSys*, 2020.

[5] J. Yu, L. Yang, N. Xu, J. Yang, and T. S. Huang, "Slimmable neural networks," in *7th International Conference on Learning Representations, ICLR*, 2019.

[6] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *23rd International Conference on Pattern Recognition, ICPR*. IEEE, 2016.

[7] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, G. Maghanath, and S. Bagchi, "Approxnet: Content and contention aware video analytics system for the edge," *CoRR*, vol. abs/1909.02068, 2019.

[8] B. Kutukcu, S. Baidya, A. Raghunathan, and S. Dey, "Contention grading and adaptive model selection for machine vision in embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[9] B. Taylor, V. S. Marco, W. Wolff, Y. Elkhatib, and Z. Wang, "Adaptive deep learning model selection on embedded systems," in *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2018*.

[10] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, pp. 55:1–55:21, 2019.

[11] A. Yazdanbakhsh, C. Angermüller, B. Akin, Y. Zhou, A. Jones, M. Hashemi, K. Swersky, S. Chatterjee, R. Narayanaswami, and J. Laudon, "Apollo: Transferable architecture exploration," *CoRR*, 2021.

[12] A. Kumar, A. Yazdanbakhsh, M. Hashemi, K. Swersky, and S. Levine, "Data-driven offline optimization for architecting hardware accelerators," in *The Tenth International Conference on Learning Representations, ICLR*, 2022.

[13] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical design space exploration," in *27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2019*. IEEE Computer Society, 2019.

[14] M. Zuluaga, G. Sergent, A. Krause, and M. Püschel, "Active learning for multi-objective optimization," in *Proceedings of the 30th International Conference on Machine Learning, ICML 2013*.

[15] I. Dagli, A. Cieslewicz, J. McClurg, and M. E. Belviranli, "Axonn: energy-aware execution of neural network inference on multi-accelerator heterogeneous socs," in *DAC '22: 59th ACM/IEEE Design Automation Conference*.

[16] E. Jeong, J. Kim, and S. Ha, "Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 5, oct 2022.

[17] D. Kang, J. Oh, J. Choi, Y. Yi, and S. Ha, "Scheduling of deep learning applications onto heterogeneous processors in an embedded device," *IEEE Access*, vol. 8, pp. 43 980–43 991, 2020.

[18] M. Parsa, J. P. Mitchell, C. D. Schuman, R. M. Patton, T. E. Potok, and K. Roy, "Bayesian multi-objective hyperparameter optimization for accurate, fast, and efficient neural network accelerator design," *Frontiers in Neuroscience*, vol. 14, 2020.

[19] M. S. Iqbal, J. Su, L. Kotthoff, and P. Jamshidi, "Flexibo: Cost-aware multi-objective optimization of deep neural networks," *CoRR*, vol. abs/2001.06588, 2020.

[20] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 6105–6114.