# Colony of NPUs: Scaling the Efficiency of Neural Accelerators

Babak Zamirai, Daya Shanker Khudia, Mehrzad Samadi, Scott Mahlke
University of Michigan, Ann Arbor, MI
{zamirai, dskhudia, mehrzads, mahlke}@umich.edu

## 1. Introduction

Trading small amounts of output accuracy to obtain significant performance improvement or reduce power consumption is a promising research direction. Many modern applications from different domains such as image and signal processing, computer vision, data mining, machine learning and speech recognition are amenable to approximate computing. In many cases, their output is interpreted by humans, so infrequent variation in outputs are not noticeable by users. Moreover, the input of these applications is noisy which prevents strict constraints on the output of the system. Hence, these domains have significant potential for the aforementioned trade-offs, and these applications are ideal targets for approximate computing.

Existing approaches for approximate computing include modifying the ISA [6], compiler [1], programming language [2, 4], underlying hardware [14] or the entire framework [3, 8, 10]. Approximation accelerators [5, 7, 13] use some of these methods to trade off accuracy for higher performance or energy savings. These accelerators require the programmer to annotate code sections which are amenable for approximation. Approximate parts are offloaded to the accelerator and other parts are executed by the CPU at run time. Esmaeilzadeh et. al. [7] proposed a Neural Processing Unit (NPU) as a programmable approximate accelerator. The key idea is to train a neural network (nn) to mimic an approximable region of original code and replace that with an efficient computation of the learned model.

Although the proposed approximate accelerators such as NPU demonstrate acceptable results on the benchmarks from different domains, they have some shortcomings.

Firstly, different invocations of a program might produce varying output qualities because the output quality is dependent on the input values. Consequently, using an NPU with fixed configuration for a wide range of inputs may produce outputs with poor accuracies. On the other hand, if the output quality drops below a determined threshold, one way to improve the quality is re-executing the whole program on the exact hardware. However, the overhead of this recovery process may offset the gains of approximation.

Secondly, existing techniques usually measure the output quality by averaging errors in individual output elements, e.g., pixels in an image. Previous works in approximate computing [9, 11, 12] show that although most of the output elements have small errors, there exist a few output elements that have considerably large errors which may degrade the whole user experience. Therefore, some inputs might need more specialized methods to deal with the issue of output quality.

Lastly, it is challenging to tune the output quality of an approximate hardware dynamically. If different NPUs with different configurations are available in the system, defining the number of active NPUs per invocation will be a reasonable knob for changing output quality based on users preferences.

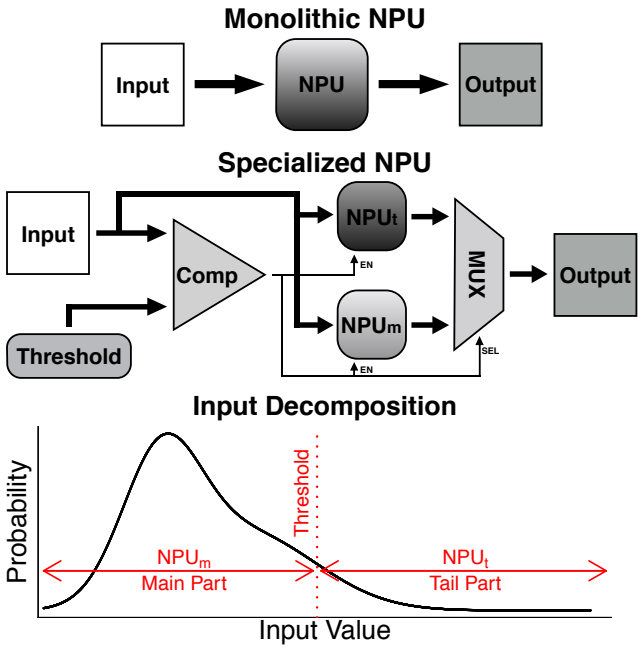


Figure 1: Employing two specialized and simple NPUs for different parts of the input range instead of a monolithic NPU.

## 2. From Monolithic to Specialized NPUs

Conventional NPUs utilize a single configuration for all ranges of all input elements. The best nn topology is produced statically by the compiler based on the training data to balance between accuracy and efficiency. In addition to the topology, the weights of each neuron are calculated during the training phase and do not change at run time for different inputs. This static NPU increases the possibility of resulting in lower-quality outputs than the average case, for some of the inputs.

The inputs of the applications which are suitable for approximate computing, such as pixels of an image, could be modeled as random variables with different distributions. For simplification, let us consider an accelerator with a single input and a single output. As depicted in Figure 1, the distribution of this input is composed of dissimilar parts, the tail part and the main part. The main part is much denser than the tail part, so it contains a lot of data points spread on a relatively limited interval. But the tail part is a wider range consists of a few data points. Using the monolithic NPU is not a good solution because each part tries to modify the network to perform more accurately for its own data. In other words, the data points in the main part might change the weights of the neurons in

a way that decreases the quality of approximation for the tail part and vice versa.

Our solution for this particular problem is finding the proper threshold for dividing the input distribution to the tail part and main part by examining the training input set. Then, we are able to divide training input set into subsets (two in Figure 1), and utilize them to train different specialized NPUs. Finally, the same mechanism is used to dispatch data to the proper NPU at run time. Each NPU is configured and trained for a limited range of data with more similar behaviors, hence the proposed system delivers the following benefits:

- More accuracy: A single network is not used for both dense and sparse data.
- Less cost: Each NPU does not need to cover a broad range of data, so $NPU_t$ and $NPU_m$ can be constructed with less number of layers and neurons per layers which decreases power consumption.
- Better performance: Smaller NPU results in lesser amount of computation, thus improving performance.

## 3. Colony of NPUs

To enhance the concept of specialized NPUs, make it more general, and address the issues discussed in the introduction section, we propose a colony of NPUs (cNPU). The goal of cNPU is to dynamically choose the best topologies and configurations based on the current input, a limited number of previous outputs and the variance of outputs of different configurations related to the last input. This specification provides a system with one or a few number of simpler working NPUs per invocation instead of a fixed complicated NPU for all invocations. Since the NPU consists of layers including fully-connected set of neurons, decreasing the number of neurons and complexity of the neural network causes better performance and energy efficiency but worse accuracy. However, we are nullifying this reduction of accuracy by increasing the similarity of input elements which makes them appropriate for accurate approximation, even for small, simple NPUs. Figure 2 shows an overview of cNPU which consists of two major components, selector and combiner.

### 3.1 Input Partitioning

Different input elements might participate in isolated computations and may have different distributions. Our goal is to partition them based on their target computation to use specific NPUs for each group. In addition, it is possible to define some thresholds on their distributions, as shown in Figure 1, to divide the range of each input element. Then, we assign specific NPUs to each subset of data. However, because we only search among simple NPUs, it is very likely that a single configuration might work for more than one part of the inputs which will reduce the number of NPUs. Another important point is that each sample of each input lies in one of its regions, so we need to run at most one NPU for each group of inputs per invocation. We can invoke multiple NPUs per input to predict error as described in next section.

### 3.2 Selector Unit

As a fixed number of layers and neurons per layer does not work properly for all kinds of inputs, we decided to utilize different neural networks with various configurations, each of which approximates output accurately for a limited range of inputs or a part of input elements. A selector is a necessary part of this framework to achieve the input partitioning goal.

The training input set could be divided to smaller sets based on two approaches or a combination of both. First, if we have n input

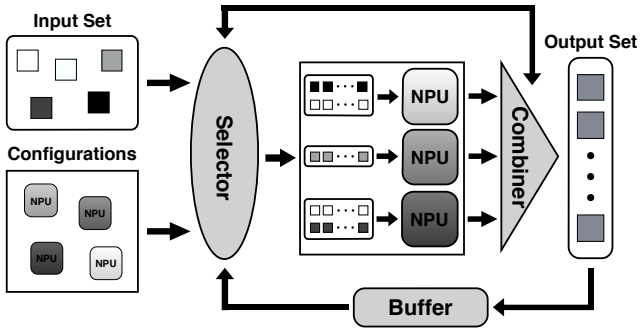


Figure 2: An overview of the cNPU system. The selector unit chooses some simple NPUs from the colony instead of the single complicated NPU based on the current input, a set of previous outputs and diversity of outputs of different NPUs for the last input. Then, the combiner unit will aggregate the outputs of the small NPUs and constructs the final output.

elements per invocation, it will be possible to form k groups in a way that each group contains the correlated elements. Second, each element has a limited range of variation, so they can be categorized by amount of each element. After forming training input sets, we choose the best NPU configuration for each group, and train that NPU by its specific input set. Then, all these NPU configurations are saved in an array in the cNPU. Limiting the range of inputs and making them more correlated in each group enables us to increase the possibility of approximating output more precisely by utilizing a simpler and smaller neural networks.

The selector unit uses a filter to choose the appropriate configurations during runtime for test inputs. Consequently, for each invocation, one or a few number of NPUs are computing the result. Three types of features are used to help the filter for choosing the best configuration. First, the runtime input itself are expected to be similar to train input, so we can apply the same filter as training phase to partition its domain. Second, a buffer is provided in system to keep record of limited number of previous outputs. They will be good clues for filtering specially if we use accurate outputs as a filtering metric during training phase. Third, because each NPU is simpler than baseline NPU, we are capable of running multiple NPUs each invocation and compare their outputs in the combiner unit and send it back to selector unit as feedback to choose the best configuration. Comparing the outputs of different NPUs gives us a hint about error. If all NPUs converge to the same results, we can predict that current configurations are working properly, else we need to substitute some of them with the idle ones.

### 3.3 Combiner Unit

This unit gets the outputs of different NPUs and computes the final result of system. Most of the time, a few number of tiny NPUs are responsible for one or a few number of output elements. Thus, each NPU is calculating the whole amount or a piece of those final elements. Different mechanisms such as weighted sum and voting can be used for integrating output segments. The selector unit passes current partitioning algorithm to combiner unit to assist this unit in selection of the best approaches for aggregating current NPU results to form the eventual outputs. Moreover, in case of using multiple NPUs, this unit is in charge of comparing the redundant results and sending feedback to the selector unit.

## References


[1] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, Mar. 2009. URL `http://hdl.handle.net/1721.1/46709`.

[2] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, pages 85 –96, 2011.

[3] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 198–209, 2010.

[4] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain¡T¿: A first-order type for uncertain data. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–66, 2014.

[5] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu. Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *Proc. of the 19th Asia and South Pacific Design Automation Conference*, pages 201–206, 2014.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, 2012.

[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 449–460, 2012.

[8] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats for software performance and health. In *ACM Sigplan Notices*, volume 45, pages 347–348. ACM, 2010.

[9] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Roomba: An online quality management system for approximate computing. page To Appear, 2015.

[10] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 213–224, 2011.

[11] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning approximation for graphics engines. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 13–24, 2013.

[12] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–50, 2014.

[13] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 1–12, 2013.

[14] T. Y. Yeh, P. Faloutsos, M. Ercegovac, S. J. Patel, and G. Reinman. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 394–406. IEEE, 2007.