Employing Classification-based Algorithms for General-Purpose Approximate Computing

Geraldo F. Oliveira[†‡], Larissa Rozales Gonçalves[†], Marcelo Brandalero[†], Antonio Carlos S. Beck[†], Luigi Carro[†]

[†]Informatics Institute – Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[‡]Computer Science Department – ETH Zürich – Zürich, Switzerland
{gfojunior,lrgoncalves,mbrandalero,caco,carro}@inf.ufrgs.br

ABSTRACT

Approximate computing has recently reemerged as a design solution for additional performance and energy improvements at the cost of output quality. In this paper, we propose using a tree-based classification algorithm as an approximation tool for general-purpose applications. We show that, without any hardware support, completely implemented in software, our approach can improve performance by up to 4x (1.95x on average) and reduce EDP by up to 19x (4.04 on average) when compared to precise executions. Besides that, in some cases, our software-based mechanism can even outperform traditional hardware-based Neural Network's state-of-the-art designs.

CCS CONCEPTS

• **Computing methodologies** → *Classification and regression trees*;

KEYWORDS

Approximate Computing; Decision Tree; Neural Networks; High Performance

ACM Reference Format:
Geraldo F. Oliveira[†‡], Larissa Rozales Gonçalves[†], Marcelo Brandalero[†], Antonio Carlos S. Beck[†], Luigi Carro[†]. 2018. Employing Classification-based Algorithms for General-Purpose Approximate Computing . In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196043

1 INTRODUCTION

The constant rise in the volume of information currently available alongside the beginning of the Dark Silicon era have been forcing computer architects to rethink the whole computer stack, seeking for both clever algorithms and specialized hardware architectures. For example, in many modern and meaningful application domains, such as computer vision, multimedia processing, machine learning, and data mining, computation is naturally tolerant to some degree of imprecision. In this context, **approximate computing** is a paradigm that has introduced quality as a new metric into the design space: the error rate. Approximate computing allows the user to obtain a simpler (ideally faster and more energy efficient) function that maps a given set of inputs to a smaller output space, including some degree of error due to this transformation. Neural Networks (NNs) have largely been used for approximate computing, and are usually implemented as dedicated hardware units to execute error-tolerant code. Although this approach is useful because of its generality, additional hardware support is a must to run the NN efficiently; otherwise, the overall system might lose performance instead of providing gains. That happens because executing NNs involves the need for computing a large number of General Matrix to Matrix Multiplication (GEMMS) in sequence over a broad set of floating-point numbers [4].

Another well-known approach for approximate computing relies on memoization [2, 13]. In regular memoization, the result of a computation with a given set of inputs is stored in memory so it can be reused next time, skipping actual execution. However, it is heavily dependent on the available memory size to store information, which increases access latency to a point where pure reuse will not pay off. One can reduce the memory footprint by mapping more than one single set of inputs to the same output, thus introducing a certain degree of imprecision. Even though it alleviates the latency problem, it is still a limiting factor for its efficient implementation, which is only possible with special hardware based memory mechanisms.

In this paper, we propose the use of an alternative learning algorithm to NNs alongside memoization, capable of keeping the learning and generic capabilities offered by NN but with a less costly implementation, so it can be implemented without any additional hardware resources (i.e., wholly in software). We do so by employing Decision Tree Learning (DTL) [22], which is a supervised learning mechanism popularly used as a classifier tool and is computationally cheaper than NN [20]. It has been applied to a broad range of applications, such as medical diagnosis and credit risk of loans. By using DTL to approximate application's kernels, we are merging the learning capabilities of the classifier algorithm used to construct the decision tree with memoization techniques to store the tree and recover it at run-time to efficiently execute the approximable kernel.

Contributions. The major contributions of this work are:

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196043

- Propose Hyperion, a totally software-based mechanism that is capable of exchanging precision for performance. With Hyperion, a decision tree replaces the most time-consuming kernels of applications that tolerate approximation.
- Evaluate our design using algorithms with different characteristics (input/output size and kernel complexity) and show in which cases it is worth using Hyperion.

Our results demonstrate that Hyperion can be up to 4x (on average 2x) faster than a non-approximated application, with an EDP reduction of up to 19x (on average 4x), without any additional hardware support. We also show that, in some cases, the totally SW-based Hyperion can even outperform state-of-the-art NN implemented in hardware.

2 RELATED WORK

Approximate computing can be implemented at distinct levels across the system stack. Recent work [25] classifies the approaches to approximate computing in three levels, *1 algorithmic*, in which the changes occur directly in the application's source code without any hardware support; *2 - architecture*, in which the algorithm is adjusted according to some hardware structure, usually with compiler support; *3 - circuit*, in which the hardware is modified. Table 1 describes a set of works that make use of approximate computing techniques, grouped into the three aforementioned classes.

In the first class, one can find works as [5, 8, 15]. Implementations include loop perforation and employing software-based NNs for kernel's approximation. The most common approach in the architectural level consists of designing NNs to execute a code region more efficiently using hardware accelerators, as in [6, 7, 17, 27]. In these works, a NN is trained to approximate code regions annotated by the programmer. The NN itself can run in dedicated hardware [6, 7], in an Graphics Processing Unit (GPU) [17], or in an Field-Programmable Gate Array (FPGA) [17]. One can also find works that use approximate memoization, which is an extension to the classic memoization scheme. In this case, similar computations are allowed to share the same entry in the lookup table, as the main approximate approach [2, 13, 23]. The primary issue of memoization technique is the size of the memory required to store the look-up table. Even though a larger memory could provide a better approximation for the application, it comes to the price of a slower access latency. In the third class, circuit-level, one can find works as [9, 14, 16], ranging from implementing adders with missing transistors [9], voltage over-scaling [16], and a cache mechanism to store memory rows with similar tag address together [14].

Table 1: Related Works and their classification.

Name	Class	Approach	HW
Misailovic, S. [15]	1.	Loop Perforation	None
Neuralizer [8]	1	Neural Networks	GPU
BenchNN [5]	1	Neural Networks	None
NPU [6]	2	Neural Networks	ASIC
BRAINIAC [7]	2	Neural Networks	ASIC
DNA [27]	2	Neural Networks	GPU
SNNAP [17]	2	Neural Networks	FPGA
Fuzzy Memoization [2]	2	Memoization	SRAM
Clumsy Value Cache [13]	2	Memoization	SRAM
Paraprox [23]	2	Memoization	SRAM
Sato, Y. [24]	2	Memoization	ASIC
IMPACT [9]	3	Custom	ASIC
Doppelgänger [14]	3	Custom	ASIC
Mohapatra, D. [16]	3	Voltage Scaling	ASIC
Hyperion	**1**	**Decision Trees**	**None**

Software based approximation techniques (i.e. algorithm level) capable of improving performance or saving energy are limited in scope (e.g.: loop perforation). More generic techniques, such as NNs, can only achieve significant gains when implemented in hardware, since they are very computationally intensive. Our mechanism bridges this gap: by using decision tree algorithms to approximate any code segment that is error-tolerant, it can be as generic as NNs, but much more efficient in performance and energy consumption. Therefore, it can be completely implemented at the algorithm level, without any sort of hardware support. To the best of our knowledge, this is the first work that makes use of decision tree algorithms to produce general-purpose approximate computing. As Section 4 shows, Hyperion improves performance of a diverse set of applications by an average factor of 2x.

3 MECHANISM

The execution flow of our mechanism is divided into three steps: 1) Code Annotation, 2) Decision Tree Training, and 3) Run-time. The first two stages are performed offline, and the last one is performed as application executes. In this section, we explain in details these three steps and also present some reasoning about employing decision trees for general-purpose approximate computing.

3.1 Code Annotation

In this step, the programmer annotates the regions of code that are tolerant to approximation in the source code. He or she will take into consideration the number of times that the target kernel will be executed, the number of inputs, and the number of outputs produced. All these metrics impact the performance improvement provided by our mechanism. Additionally, the programmer provides sample inputs to the application, which will be used in the training step. Code Annotation is the only step where human intervention is required. This approach is standard practice in the literature [6, 7].

The selected code region subject to approximation must fit the *pure* definition [11], as follows:

- The number of inputs and outputs (code variables) of the region must be fixed and known at compile time;
- The region must not cause any side-effects, (*i.e.* modify the program state outside of the current region);
- The region must be deterministic, (*i.e.*, produce the same output whenever it is executed with the same input).

For example, the code presented in the Listing 1 shows a function[1] with six inputs (both *p and *c1 store three distinct elements) and one output. This code represents 34% of the total application execution time, and it is called 60%. It is also deterministic and does

[1] Our mechanism can approximate both whole functions as well code regions.

not cause any side-effect to the application, making it suitable for our mechanism.

Listing 1: Original Code with six inputs and one output

```
float euclideanDistance(RgbPixel* p,
    Centroid* c1){
    float r = 0;
    r += (p->r - c1->r) * (p->r - c1->r);
    r += (p->g - c1->g) * (p->g - c1->g);
    r += (p->b - c1->b) * (p->b - c1->b);
    return sqrt(r);
}
```

3.2 Training

To generate the decision tree that will be used during run-time, we make use of the Weka Tool [10]. Weka is a collection of machine learning algorithms implemented in Java that target data mining applications. Each kernel output will create a distinct decision tree. For that, the user must provide an input set of data that will be used by Weka to learn an appropriate decision model. This set of data is comprised of input and output elements for kernel's application being approximated. Finding the optimum tree that produces a model without output errors is a NP-hard problem. Therefore, different heuristic algorithms are applied to build the tree. Then, the kernel to be approximated will be replaced by a searching algorithm that travels through the generated trees.

The decision tree algorithm is a derivation of the popular C4.5 algorithm proposed by [19]. The input set of the algorithm is organized into attributes, classes, and a training set that relates the attributes with the classes. In our mechanism, each kernel's input is classified as an attribute, while its outputs are the classes. The algorithm produces a decision tree, where each node of the tree corresponds to an attribute, and each leaf represents a predicted class. The algorithm builds the tree from top to bottom in a greedy way, computing the information gain for each attribute. This value is a measure of the difference in entropy with and without the selected attribute in the training set. In other words, the information gain tells which attribute is the most relevant in the training set. This attribute is placed higher in the tree. Thus the searching space can be split logarithmically in each level of the tree. The attribute that gives the best gain is chosen to be the current node in the tree, and then a *split* happens, where a sub-tree with the remaining attributes is created. The process continues on the remaining attributes until all attributes have been evaluated. The algorithm can deal with both discrete and continues attributes. In the former, the attribute itself is select as the current node. In the latter, a threshold (or split value) is computed to integrate the node, and then both the threshold and the attribute are stored in the node. For a more detailed description of the algorithm, we refer the reader to [3, 21].

3.3 Running

First, when the code starts running, the application loads the file created by the previous step into a tree data structure. When the approximated kernel is invoked, a simple binary search is executed in place of the original computation. Listing 2 shows the code that will be executed when the function *euclidianDistance* is called. Since this function has one output, only one tree will be required. Starting from the root of the tree, the algorithm tests if one of the six inputs (*p and *c1 are stored in the *dataIn* array of six positions) is less than the threshold value (the same as the *split_value* obtained during the training phase). The *position* value stored in the node of the tree corresponds to an attribute of the training set. If less than the threshold, the searching process continues on the left sub-tree, otherwise, it goes to the right sub-tree. This process is repeated until a leaf is reached. The returned value is the one stored on the reached leaf.

Listing 2: Modified code with searching algorithm.

```
float euclideanDistance(RgbPixel* p,
    Centroid* c1){
    dectree_node *search;
    search = open("tree");
    float input , threshold;
    while(!search -> is_leaf){
        input=dataIn[search -> position];
        threshold = search -> split_value;
        if( input < threshold)
            search = search -> left;
        else
            search = search -> right;
    }
    return  search -> final_value;
}
```

3.4 Decision Trees for General-Purpose Approximation

Since decision trees are not usually employed as a learning mechanism for general-purpose kernels, but instead for classification over data sets, we needed to evaluate how it would adapt to the former. For this end, we analyzed the distribution of the output values produced when approximating the *inversek2j* algorithm using our decision tree-based mechanism via the histogram illustrated in Figure 1. By employing the classification algorithm, we can reduce the size of the output space produced by the application. It is important to notice that the outputs produced via approximation still lie within the non-approximated one. The conclusion we can take from these observations is twofold. First, by reducing the size of the output space, fewer operations will be required to be performed by the application, then improving performance. Second, since the output values approximated by the decision tree algorithm are within the original output space, the error induced by making the approximation itself might be acceptable by the application[2]. On top of that, by storing the reduced output space in a tree-based data structure, we can potentially reduce the complexity of the original computation to $O(\log n)$, since now the primary operation being executed is searching over a binary tree.

4 EXPERIMENTAL SETUP AND RESULTS

In this section, we evaluate Hyperion regarding performance, quality, and Energy-Delay Product (EDP). We have also compared our

[2]In Section 4 we discuss how one might reduce the error-rate produced by the learning algorithm.

Table 2: List of benchmarks evaluated, characteristics of each function, setup of decision trees trained, and NPU Topology.

	Description	Type	Evaluation Input Set	Training Input Set	Error Metric	Approximated Kernel	% HotSpot Calls	Avg. Tree Size	NPU Topology
blackscholes	Mathematical model of a financial market	Financial Analysis	48,000 options	48,000 options	Average Relative Error	bs_thread	51	865	7, 9, 9, 2
fft	Radix-2 Cooley-Tukey Fast Fourier Inverse	Signal Processing	32768 Random Floating Point Numbers	2018 Random Floating Point Numbers	Average Relative Error	fftSinCos	75	89	2, 5, 5, 3
inversek2j	Kinematics for 2-joint ARM	Robotics	1000000(x,y) Random Coordinates	100000(x,y) Random Coordinates	Average Relative Error	inversek2j	50	86	3, 5, 3
jmeint	Triangle Intersection Detection	3D Gaming	1000000 Random Pairs of 3D Triangle Coordinates	100000 Random Pairs of 3D Triangle Coordinates	Miss Rate	tri_tri_intersect	43	15	19, 3, 3
jpeg	JPEG Encoding	Compression	33 220x200 Pixel Color Images	Ten 512x512 Pixel Color Images	Image Diff	encodeMcu	33	68	65, 5, 65
kmeans	K-means Clustering	Machine Learning	33 220x200 Pixel Color Images	One 512x512 Pixel Color Image	Image Diff	euclideanDistance	33	15	7, 9, 2
sobel	Sobel Edge Detector	Image Processing	33 220x200 Pixel Color Images	One 512x512 Pixel Color Image	Image Diff	sobel	17	391	10, 9, 5, 2

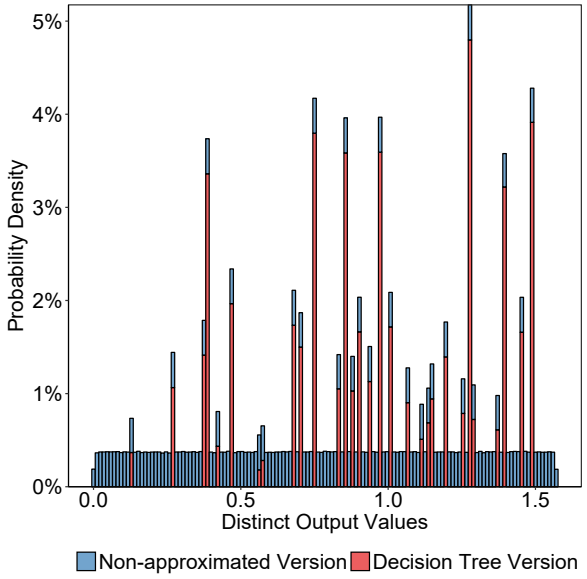

Figure 1: Histogram for the output produced by the inversek2j algorithm for both its approximate and non-approximated versions.

mechanism with a state-of-art Neural Network (NN), called NPU [6], the current baseline in approximate computing. We have chosen this particular NN because the authors of NPU made available both their software NN implementation and the benchmarks already modified for approximation. Besides that, similar to our mechanism, NPU targets general purpose applications. We also compare one of our benchmarks with Paraprox [23]. Unfortunately, the remaining benchmarks evaluated by the authors are not available.

Table 3: Hardware setup used to execute the applications.

Architecture	x86-64
CPUs	4
Model name	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
L1 Instruction Cache	32KB
L1 Data Cache	32KB
L2 Data Cache	256KB
L3 Data Cache	6MB
Main Memory	16 GB DDR3

We considered all seven applications from the AxBench suite [26], covering a significant range of domains that are tolerant to approximation, such as signal processing, games, machine learning and image processing. These applications are shown in Table 2. In this table, we also describe the approximated kernel in each application and how representative it is (in number of calls). We have chosen to approximate the same portions of code that [6], thus we could directly compare the already trained NNs provided by the authors against our mechanism. Finally, Table 3 describes the CPU setup used to run our experiments. We evaluated performance in a real x86 processor with the PAPI API [18] (for both our mechanism and NPU software implementation), while energy was measured using Intel RAPL Power Meter.

Figure 2 shows the speedup for each application when comparing against the non-approximated algorithm, and also Hyperion's geomean speedup. We also plot the speedup achieved by Paraprox [23] for the *blackscholes* benchmark and by the NPU mechanism using both a hardware accelerator unit[3] and its software implementation. It is important to point out that both Paraprox and NPU HW take advantage of specialized hardware units, while our mechanism runs completely in software. First, when analyzing the

[3]The speedup values were extracted directly from [6].

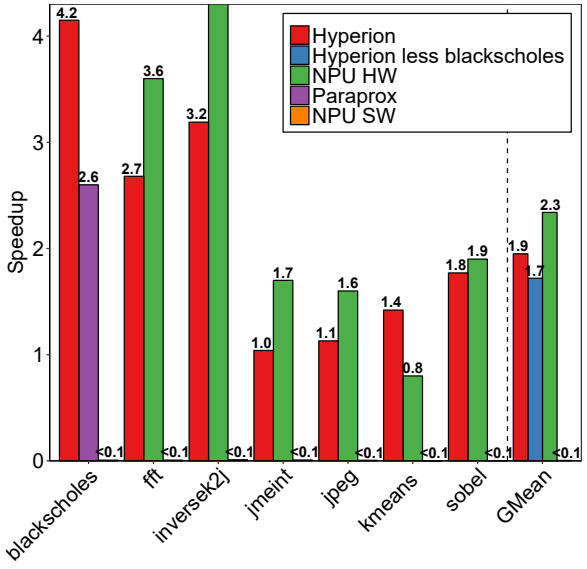

Figure 2: Speedup obtained when comparing the baseline and our mechanism.

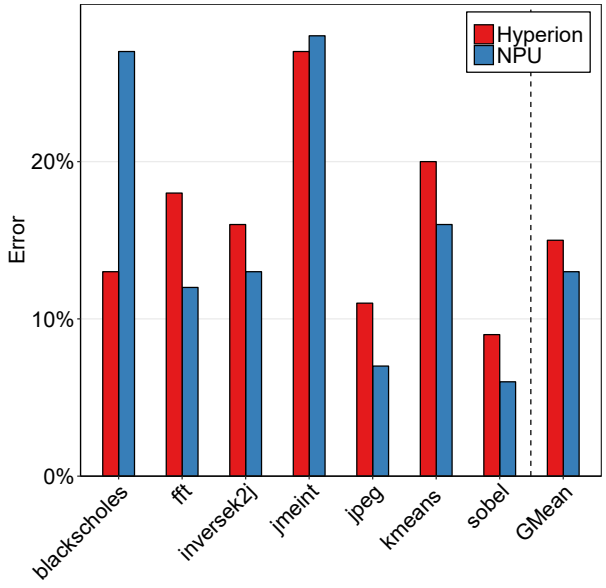

Figure 3: Error produced by Hyperion and NPU when comparing with the non-approximated algorithm.

speedup produced by Hyperion, one can notice that our best result is for the *blackscholes* benchmark (4.15x), while the minimum speedup obtained was for *jmeint* (1.04x). To understand our results, we have profiled each application regarding the type of executed instructions. Our profile has shown that by using our tree-based mechanism we were able to reduce by 7x the number of floating-point operations for the *blackscholes*, while keeping the number of ALU and load/store operations almost constant (1.03x, 1.2x, 1.x, respectively). Meanwhile, the number of floating-point operations was reduced by 4x for the *jmeint*. Even though this reduction is significant, the L1 and L2 data cache miss rate increased by almost two orders of magnitude, which explains the small acceleration in this case.

Also, in the same Figure, one can notice that we outperform the NPU software implementation. The authors of NPU had already discussed in [6] the inability to employ NNs in software, due to the massive number of General Matrix Multiply (GEMM) computation required to evaluate each neuron in the network. When comparing against their hardware NN accelerator, we could obtain better performance for the *kmeans* benchmark, even with a purely SW based technique. We can achieve an average speedup closer to the NPU hardware implementation (Hyperion can obtain an average speedup of 1.72x when not considering the *blackscholes* benchmark[4], while NPU can achieve a speedup of 2.3x). When comparing our decision-tree implementation of the *blackscholes* algorithm with the memoization implementation proposed by Paraprox, we obtain almost 2x their speedup without any additional cost in area (as aforementioned, Paraprox makes use of additional memory units to perform its memoization mechanism). That shows that including a more sophisticated learning mechanism alongside memoization can provide significant performance improvements.

[4]We have removed the *blackscholes* benchmark from the calculation because the authors of NPU did not employ this benchmark in their work.

Figure 3 depicts the error induced by our mechanism. In general, our approach results in slight higher error rates than the NN approach (an average error of 15.2 % against 13.4 % produced by the NN). In some particular applications (*blackscholes* and *jmeint*), we can obtain less error than the classic NN approach. However, on average, our mechanism does not add more than 4 % of error when comparing with all NPU implementations.

We also have investigated the impact of increasing the decision trees with the C4.5 training algorithm on the error rate. For that, we have trained all applications increasing the size of the tree at each iteration until the error-rate stabilized or a certain number of iterations (1000) was reached. The only benchmarks that showed a better tree-implementation were *fft* and *kmeans*. For the former, the average size of the tree went from 89 nodes to 94. For the second one, the number increased from 15 nodes to 22. However, the impact on the error factor was less than 1 % for each benchmark.

Finally, Figure 4 shows the reduction in EDP Hyperion can produce. For the *blackscholes* benchmark, we can achieve an EDP of 19x better when compared with its non-approximated counterpart. On average, our mechanism can obtain an EDP reduction of around 4x. Even though Hyperion has presented meaningful results, there is still room for improvements. The current implementation basically does pointer-chasing operations, which makes memory access to become too sparse, harming data locality. Moreover, since the trained tree is stored alongside application's data, it shares hardware resources with the user application. This potentially increases the number of accesses to the main memory, which is around 1000x more costly than float-point operations [1, 12]. Moreover, transversing the three will affect branch prediction and other prefetching mechanisms. However, both issues can potentially be solved: previous works [1, 12] have already proposed solutions for such algorithms targeting improvements on locality.

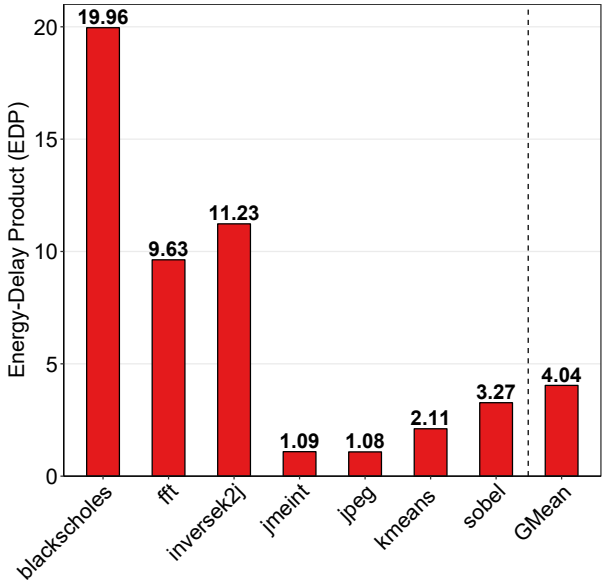

Figure 4: EDP reduction produced by Hyperion when comparing with the non-approximated algorithm.

5 CONCLUSIONS AND FUTURE WORK

In this work, we presented Hyperion - a software technique that uses decision trees to approximate applications' outputs. The mechanism transforms the execution of an approximable code region into a simple binary tree search that selects the best-saved output for the requested input. We have shown that our mechanism can obtain significative performance (up to 4x and an average of 1.95x) and Energy-Delay Product (EDP) reduction (up to 19x and an average of 4x) for applications that calculate complex operations, as floating-point computation, while keeping the error produced by our mechanism closer to the one produced by popular state-of-the-art Neural Network (NN) implementation.

Moreover, we have laid the base for identifying when the proposed software technique can be applied, and hence as future work, we will create a complete framework at the compiler level to facilitate employing our mechanism. Besides that, we expect to design an accelerator unit, with low area and energy costs, to accelerate our mechanism.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 105–117.

[2] Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.* 54, 7 (2005), 922–927.

[3] David Bayu Ananda and Ari Wibisono. 2014. C4. 5 Decision Tree Implementation In Sistem Informasi Zakat (Sizakat) To Automatically Determining The Amount Of Zakat Received By Mustahik. *Jurnal Sistem Informasi* 10, 1 (2014), 28–35.

[4] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2017. Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes. *arXiv preprint arXiv:1701.06420* (2017).

[5] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michele Sebag, and Olivier Temam. 2012. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 36–45.

[6] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 449–460. https://doi.org/10.1109/MICRO.2012.48

[7] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 615–626.

[8] Beayna Grigorian and Glenn Reinman. 2015. Accelerating Divergent Applications on SIMD Architectures Using Neural Networks. *ACM Trans. Archit. Code Optim.* 12, 1, Article 2 (March 2015), 23 pages. https://doi.org/10.1145/2717311

[9] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. 2011. IMPACT: Imprecise Adders for Low-power Approximate Computing. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design (ISLPED '11)*. IEEE Press, Piscataway, NJ, USA, 409–414.

[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[11] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 1–6.

[12] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE, 25–32.

[13] Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. 2015. Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders. In *1st Workshop On Approximate Computing (WAPCO 2015)*. 6.

[14] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 50–61.

[15] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of Service Profiling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/1806799.1806808

[16] Debabrata Mohapatra, Vinay K Chippa, Anand Raghunathan, and Kaushik Roy. 2011. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 1–6.

[17] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 603–614. https://doi.org/10.1109/HPCA.2015.7056066

[18] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[19] J Ross Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.

[20] Muhammad A Razi and Kuriakose Athappilly. 2005. A comparative predictive analysis of neural networks (NNs), nonlinear regression and classification and regression tree (CART) models. *Expert Systems with Applications* 29, 1 (2005), 65–74.

[21] Salvatore Ruggieri. 2002. Efficient C4. 5 [classification algorithm]. *IEEE transactions on knowledge and data engineering* 14, 2 (2002), 438–444.

[22] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.

[23] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 35–50.

[24] Yuuki Sato, Takanori Tsumura, Tomoaki Tsumura, and Yasuhiko Nakashima. 2015. An Approximate Computing Stack Based on Computation Reuse. In *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 378–384. https://doi.org/10.1109/CANDAR.2015.35

[25] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2016. Approximate computing: A survey. *IEEE Design & Test* 33, 1 (2016), 8–22.

[26] Amir Yazdanbakhsh, Divya Mahajan, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2016. *AxBench: A Benchmark Suite for Approximate Computing Across the System Stack*. Technical Report. Georgia Institute of Technology.

[27] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural Acceleration for GPU Throughput Processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 482–493. https://doi.org/10.1145/2830772.2830810