History-based Piecewise Approximation Scheme for Procedures

Aurangzeb and Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
orangzeb@purdue.edu, eigenman@purdue.edu

***Abstract*—Approximate computing has emerged as an active area of research in recent years. A number of techniques have been proposed to increase performance of applications that can tolerate a certain degree of inaccuracy. Approximating functions can offer significant performance improvement but this opportunity has not yet been fully explored by the approximate computing community. This paper introduces techniques to perform approximation at the level of functions. We present our schemes with two flavors and discuss four realizations. We present results on 90 scientific functions that underscore the opportunity. We also present results on real applications that demonstrate the practicality and effectiveness of the idea.**

***Keywords*-approximate computing; function approximation; history-based piecewise approximation**

I. Introduction

The large majority of today's computer applications produce exact and reproducible answers. However, not all applications require 100% accuracy. Many applications in audio, video, image processing, gaming, information retrieval and analysis, and machine learning can tolerate inaccuracies. Such applications have significant potential for performance improvement in exchange for accuracy. This opportunity has led to the emergence of the field of approximate computing. The present paper explores such approximation at the granularity of functions and provides evidence of the potential gains.

Function approximation has been pursued in different well-established fields of Mathematics and Computers such as Numerical Analysis and Machine Learning and a number of techniques have been proposed. Among the numerical analysis techniques, cubic splines and Chebyshev polynomials are generally considered the best function approximation schemes. Machine learning techniques such as Artificial Neural Networks (ANNs), Support Vector Machines (SVMs), and fitness approximation in Genetic Algorithms (GA) provide good approximation, but their software implementations are slow and only benefit complex applications and compute-intensive simulations. The approximate computing community has explored approximation at the levels of hardware, architecture, data types, instructions, loops, and synchronization blocks, but not so much at the level of procedures. Two attempts related to function approximation are a neural network approach [1] and an approximate memoization technique [2]. The former is shown to be effective only when implemented in hardware and the latter is described as a pattern-based approximation scheme applicable to map pattern in data-parallel applications and only targets GPU applications. By contrast, our schemes target general applications and do not require customized hardware.

This paper makes the following contributions:

1) We introduce a function approximation scheme that we call *history-based piecewise approximation* and explore several options.
2) We show that mathematical and scientific functions can be approximated and used in applications amenable to approximate computing by presenting results on 90 such functions from the GNU Scientific Library (GSL) [3], [4]. Results show that our approximation scheme was able to speed up 92% of all functions. For 71% of the functions, the normalized RMS error in the approximated result was very small (0.06 on average) with 9.3x speedup, on average. Whereas for another 15% of functions it was 0.49 with an average speedup of 9.5x.
3) We demonstrate the feasibility and practicality of the approach by presenting results on three real applications. The average speedup for these applications is 1.74x with 0.5% error, on average.

The rest of the paper is organized as follows. In Section II, we introduce and describe a *history-based piecewise approximation scheme* with two flavors and their realizations. In Section III, we present results of our experiments with mathematical functions and applications. We conclude in Section VI after presenting future work in Section IV and related work in Section V.

II. History-based Piecewise Approximation

Polynomial approximation is one of the well-known Numerical Analysis techniques for function approximation. Theoretically, according to the Weierstrass theorem, any continuous function can be approximated well by a polynomial. However, the kind and degree of polynomial as well as the data points used to find the coefficients of the polynomial impact the approximation and are difficult to determine in practice. Moreover, approximating a function using a single polynomial for the entire input range of the function does not produce good results. Piecewise approximation, in which the input range of the function is split into different regions and a polynomial is used for approximation in each region, yields better approximation results. However, deciding on how to split the input range and which polynomials to use are again difficult choices to make. We introduce a piecewise approximation scheme in which the regions are formed based on past history of the function invocations and lower order polynomials are used for approximation; we call it history-based piecewise approximation. We introduce two sub-schemes, non-uniform piecewise approximation, and uniform piecewise approximation.

During a training phase, a history of input and output values of the original function is recorded. There are different scenarios for training, such as offline, static online (at the beginning of the application execution) and dynamic online training. The history is then used to form regions for piecewise approximation and compute the coefficients of the polynomials used for approximating each region. In the production phase, for each function input, a search is performed on the history to find the region that the input falls into. The corresponding polynomial for that region is then used to compute the output. The history, lookup mechanism, and the polynomials used affect the speed of approximation and thus the overall performance.

A. History-based Non-uniform Piecewise Approximation

Non-uniform piecewise approximation forms regions that are of non-uniform size. Formation of these regions is strictly dictated by the learnt history of function invocation during the training phase, which for this scheme is currently performed in the beginning of the application execution. The non-uniform nature of the scheme allows non-uniform concentration of history elements, which can improve approximations for hot/dense input regions. The current approximation strategy uses either a 0-degree (a constant) or a 1-degree (a straight line) polynomial but higher-order polynomials can also be used.

We have realized this scheme in three different ways depending on the underlying storage and lookup mechanism.

1) Binary Search on Sorted Array: This method allows recording input-output values of the function during training in the order they appear. The array is then sorted based on input values using Quick Sort at the end of the training phase. During the production phase, binary search is performed on the sorted array to find the region an input falls into. Since the inputs are sorted, it is easy to find the corresponding region. The size of the array depends on the length of training; the evaluation section reports on different options. One limitation of this scheme is that it does not allow the history to grow once the first production phase has started and thus it cannot support dynamic training.

2) Binary Search Tree: This realization uses a Binary Search Tree for storage and lookup. During training, input-output pairs are stored as nodes in the tree. These history elements are added in the tree as they arrive. Inserting a node maintains the binary search tree property: for any node with input value **v**, all input values in the left sub-tree are less than **v**, and all input values in the right sub-tree are greater than **v**. Because this property is maintained during insertion, no extra sorting is needed at the end of training. This implementation allows the history to grow dynamically during the application run. It also supports non-uniform concentration of history elements.

3) Red-Black Tree: For general cases, binary search trees offer a reasonable storage and lookup mechanism, but their structure is dependent on the sequence in which history elements are added. In the worst case, if the history elements during training appear in a way that their input values are in ascending/descending order, a binary search tree can end up being a linear list and require O(N) accesses for a search. To avoid this worst-case scenario, the third realization uses red-black trees. Red-Black trees are approximately balanced. Balance is achieved using an extra red or black color attribute for all nodes and enforcing the red-black property in addition to the binary search tree property during insertion. The red-black property is as follows: every node is either red or black, the root and all leaves are black, if a node is red then both children are black, and for each node, all paths from that node to descendant leaves contain the same number of black nodes. The height of a red-black tree with **n** internal nodes is at most 2log(**n**+1). We used the algorithms mentioned in [5] for insertion of nodes.

B. History-based Uniform Piecewise Approximation

In this scheme, the entire range of input values is split into uniform regions. This range is found through the history either via a profile run or by guessing based on the first **m** invocations of the function. This scheme does not allow non-uniform concentration of history elements, which may negatively affect the approximation results in the hot/dense regions of the input if there are a lot of variations in the output in those regions and the size of uniform regions is not small enough to be able to capture them.

We realized our uniform piecewise approximation scheme using a hash-table. The overall range of input values for a function is uniformly split into different regions and mapped to the hash-table such that the inputs in different regions hash to different buckets in the hash-table. The hashing function ensures this mapping. Using hash-based storage and lookup offers a performance benefit, but does not support dynamic expansion. Our current scheme finds the overall range of input through profiling. This scheme uses a slightly different training mechanism than the non-uniform scheme. The buckets in the hash-table are initialized to be in an untrained state. The first input value in a region, whenever it appears during the application execution, serves to train the polynomial for the region it falls into. For later input values in that region, approximation is performed by evaluating the polynomial whose coefficients were determined during the training. In the current version, one history value is stored per region and a constant (0-degree polynomial) is used for approximation. The same scheme can support other variants, such that multiple values are stored for each region and higher-order polynomials are employed to approximate results.

Discussion

Table I summarizes the properties of all these techniques. Tree-based schemes naturally support extensible history and thus dynamic training. They also support non-uniform concentrations of input values so that they are able to better approximate against an input value that falls in the dense region of input. However, they are slower as compared to the uniform scheme, which on the other hand, does not allow the history to grow and does not support non-uniform concentration of inputs. These techniques complement each other and could be combined in an overall piecewise approximation scheme.

III. Evaluation

A. Results for 90 Functions from GSL

Below we present our results for many mathematical and scientific functions from GSL (GNU Scientific Library) [3], [4], which is a widely used open source numerical library for C/C++, distributed under the GNU General Public License. Table II summarizes the results of approximating functions using our history-based uniform piecewise scheme. Testing is performed on a machine with Intel Core2 Duo CPU running at 3GHz, 6144 KB cache and 4GB of RAM, running Ubuntu 12.04. For each function, we report speedup against the RMS error in the results due to approximation. The reported RMS is normalized by the average actual output of the function for the tested inputs. We also describe the overall range of the input used in the experiment for each function and the size of the region in our uniform piecewise approximation scheme. We split the input range into 5000 regions. We called each function 1,000,000 times with different random inputs in the given input range. This number is realistic in real applications as can be seen in column four of Table III. How much error for a given function can be tolerated, would depend on the applications, but results show that most of the functions are successfully approximated with an RMS error that may be tolerable by many applications. Out of a total of 90 functions, 71% report a normalized RMS error of 0.06, on average, for an average speedup of 9.3x. 15% report a normalized RMS error of 0.49, on average, for an average speedup of 9.5x. The remaining 14% report a large error (188.44) for a speedup of 12x. The scheme was able to speedup 92% of all functions.

B. Results for Applications

We have evaluated our approximation schemes on two applications, Blackscholes and Swaptions, from the Parsec Benchmark Suite [6] and a Convolutional Neural Network (CNN) application for handwritten digit detection (CNN-HDD). Table III presents results using the uniform scheme. For each application, the function that was approximated is listed. We determined the upper bound for the speedup each application can achieve through function approximation by temporarily altering the original function such that it immediately returns the supplied input as a result. For the CNN-HDD application, our error metric is the percentage of images that were not detected by the application, for others it is the RMS error normalized by the average output. CNN-HDD resulted in a speedup of 1.68x and detected 9897 images out of 9899 (0.02% images could not be detected by the approximated application). For Blackscholes, the speedup is 2.27x against an RMS error of 0.048. For Swaptions, the speedup is 1.28x and the RMS error is 0.005. The average speedup of applications is 1.74x. To find the average error across applications we calculated the percentage error for Blackscholes and Swaptions, which is 1.03% and 0.6% respectively. The average percentage error for three applications is 0.5%. We used the uniform piecewise approximation to obtain these results. For CNN-HDD, the scheme used 702 regions of size 0.05, for Blackscholes, 7200 of size 0.005, and for Swaptions, 200 of size 0.005.

C. Comparison of Techniques and Effect of Training

Figure 1 compares the non-uniform techniques in terms of speedup and normalized RMS error for the Blackscholes application. The binary search technique turns out to be competitive when the history is small, but for a large history it is taking more time to sort the array, whereas the scheme using red-black trees performs the best among non-uniform schemes. This figure also depicts the effect of training on speedup and error. Forming more regions helps better approximate at the cost of some decrease in speedup. In terms of approximation results, the three approaches give similar results. This is because their non-uniform regions are formed by the training history, which is the same for all schemes. Figure 2 shows the results of the uniform scheme used to test Blackscholes. Comparing figures 1 and 2, the non-uniform schemes give better approximation results but they are slower than the uniform scheme.

D. Comparison with State of the Art

We compared our schemes with the best known Numerical Analysis techniques, namely, polynomial approximation, Chebyshev polynomial approximation, and cubic splines. We used monomial basis (1, x, x^2, x^3, ...) for polynomial approximation. Chebyshev approximation chooses its own data-points and for polynomial approximation using monomials and cubic spline we tried two options - the first

Table I
PROPERTIES OF HISTORY-BASED PIECEWISE APPROXIMATION SCHEMES

Scheme	Storage & Lookup	Fixed/Extendable	Lookup Complexity	Supports Non-uniform Concentration?	Order
Non-uniform	Array + Binary search	Fixed	O(log N)	Yes	Sorted
Non-uniform	Binary search tree (BST)	Extendable	>O(log N)	Yes	Binary search tree property
Non-uniform	Red-black tree	Extendable	~O(log N)	Yes	BST + red-black property
Uniform	Hash-table	Fixed	O(1)	No	Ranges are sorted

Table II
RESULTS OF APPLYING HISTORY-BASED UNIFORM PIECEWISE APPROXIMATION ON FUNCTIONS FROM GSL

Input Range	Region Size	Function	RMS Error	Speedup	Function	RMS Error	Speedup	Function	RMS Error	Speedup
-1000.0 - 1000.0	0.4	sf_bessel_j2	0.4835	9.04	sf_legendre_P2	0.0006	0.39	asinh	0.0011	2.55
		sf_bessel_i0_scaled	0.8935	0.50	sf_legendre_P3	0.0009	0.43	sf_sin	0.1805	4.20
		sf_dawson	1.0373	7.49	sf_clausen	0.2122	6.53	sf_cos	0.1801	3.95
		sf_bessel_i1_scaled	0.3003	5.09	sf_bessel_J0	0.2491	16.44	sf_Si	0.0044	18.38
		sf_bessel_i2_scaled	0.0784	5.12	sf_bessel_J1	0.2477	17.93	sf_dilog	0.0007	7.30
		sf_atanint	0.0009	6.42	sf_bessel_j0	0.6977	4.91	sf_lncosh	0.0003	0.46
		sf_psi_1piy	0.0011	2.16	sf_bessel_j1	0.6559	8.97			
0.0 - 1000.0	0.2	sf_bessel_y2	697.93	9.52	sf_bessel_I0_scaled	0.0554	6.46	log1p	0.0004	2.30
		sf_bessel_k0_scaled	60.843	0.51	sf_bessel_I1_scaled	0.0227	6.39	sf_debye_1	0.0737	3.97
		sf_bessel_k1_scaled	491.51	0.72	sf_fermi_dirac_2	0.0004	2.05	sf_debye_2	0.3188	4.57
		sf_fermi_dirac_0	0.0002	7.08	sf_fermi_dirac_3half	0.0004	38.91	sf_log	0.0023	2.38
		sf_fermi_dirac_1	0.0003	3.32	sf_bessel_Y0	0.2992	19.95	sf_log_abs	0.0023	2.44
		sf_fermi_dirac_mhalf	0.0002	37.35	sf_bessel_Y1	14.541	21.42	sf_psi	0.1363	6.57
		sf_fermi_dirac_half	0.0002	38.71	sf_bessel_K0_scaled	0.1470	4.82	sf_lnsinh	0.0002	0.58
		sf_lambert_W0	0.0004	26.70	sf_bessel_K1_scaled	9.5141	4.83	sf_psi_1	494.29	53.86
		sf_lambert_Wm1	0.0004	27.68	sf_bessel_y0	78.269	7.94	sf_Ci	2.6739	22.50
		sf_log_1plusx	0.0004	2.45	sf_log_1plusx_mx	0.0002	2.48	sf_bessel_y1	496.56	10.17
-200.0 - 200.0	0.08	sf_bessel_I0	0.3251	8.57	sf_expint_E1	0.4165	9.56	sf_Chi	0.3239	18.58
		sf_bessel_I1	0.3251	8.41	sf_exprel_2	0.4909	3.52	sf_exp	0.4986	3.10
		sf_expint_Ei	0.4947	9.87	sf_expint_E2	0.4165	11.04	sf_exprel	0.4947	3.44
		sf_expm1	0.4986	3.26	sf_Shi	0.3239	18.45			
0.0 - 400.0	0.08	sf_debye_3	0.1174	6.59						
0.001 - 200.0	0.04	sf_bessel_K0	1.7186	8.72	sf_bessel_K1	99.966	8.84			
-50.0 - 50.0	0.02	cdf_ugaussian_P	0.0009	3.65	sf_erf_Q	0.0009	6.49	sf_erf	0.0010	6.70
		cdf_ugaussian_Q	0.0009	3.67	ran_ugaussian_pdf	0.0303	3.72	sf_erfc	0.0010	6.10
0.0 - 50.0	0.01	sf_fermi_dirac_m1	0.0002	4.13	sf_transport_2	0.0003	10.39	sf_expint_3	0.0005	1.02
		sf_lngamma	0.0003	7.48	sf_transport_3	0.0002	10.93	sf_gamma	0.1681	12.86
		sf_gammastar	0.0700	6.16	sf_transport_4	0.0002	11.43	sf_eta	0.0001	21.49
		sf_gammainv	0.0123	14.37	sf_transport_5	0.0002	11.20			
0.0 - 100.0	0.02	sf_debye_4	0.0171	12.12	sf_synchrotron_2	0.0837	7.87	sf_debye_6	0.0198	12.40
		sf_debye_5	0.0187	12.71	sf_synchrotron_1	0.1049	8.95			
-100.0 - 100.0	0.04	expm1	0.1736	3.04						
-100.0 - 1000.0	0.22	sf_log_erfc	0.0003	3.81						
-150.0 - 0.0	0.03	sf_zetam1	0.6295	40.20						
-35.0 - 35.0	0.014	sf_erf_Z	0.0179	3.38						

Table III
APPLICATIONS AND RESULTS FOR APPROXIMATION USING THE UNIFORM PIECEWISE SCHEME

Application	Domain	Function	# Invocations	Max Possible Speedup	Error	Error Metric	Application Speedup	# Regions, Region Size, Memory (KB)
CNN-HDD	Machine Learning	tanh	8,010,000	1.8	0.02%	%undetected images	1.68	702, 0.05, 5.6
Blackscholes	Financial	CNDF	13,107,200	2.48	0.048	Normalized RMS error	2.27	7200, 0.005, 57.6
Swaptions	Financial	CumNormalInv	76,800,000	1.33	0.005	Normalized RMS error	1.28	200, 0.005, 1.6

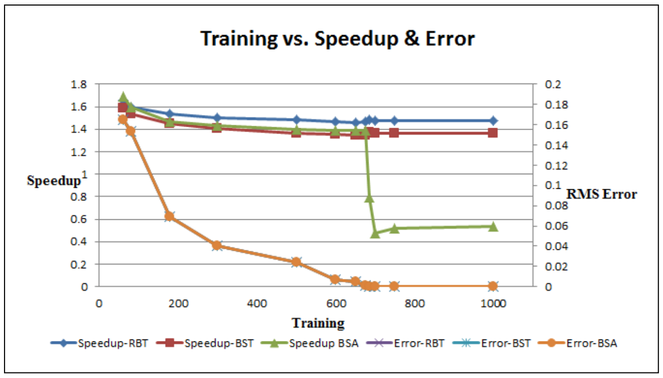

Figure 1. Comparison of non-uniform techniques for Blackscholes application and effect of training on speedup and error.

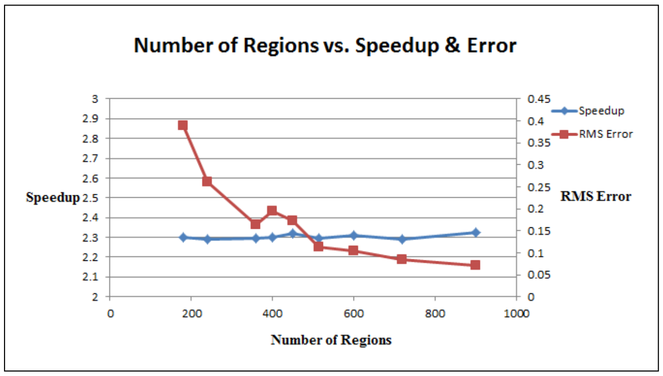

Figure 2. Uniform scheme for Blackscholes application.

n+1 inputs and evenly spaced data-points over the entire input range of the function. The approximation results of the latter schemes with the first option (first n+1 inputs as data-points) were poor (100% - 91% error). But for evenly spaced data-points they were good and are reported in figures 3 and 4. To compare the schemes, we picked the best-case for each, considering both speedup and percentage error. Cubic splines and non-uniform techniques can result in better approximation than the uniform scheme with a much less speedup but the best case we picked for these schemes also considered the speedup. Figure 3 compares the speedup of all techniques for their best-case; whereas figure 4 compares the percentage error for the best-case of each of the techniques for the CNN-HDD application. The best-cases of our schemes outperform those of numerical analysis techniques in terms of both speedup and percentage error.

IV. Future Work

The presented work has laid foundations and demonstrated the potential of approximation at the level of procedures. In future work, we will support dynamic training in our piecewise schemes and come up with an efficient and automated way of choosing techniques and their parameters. We will also combine the proposed complementary schemes in an overall piecewise approximation scheme. Furthermore,

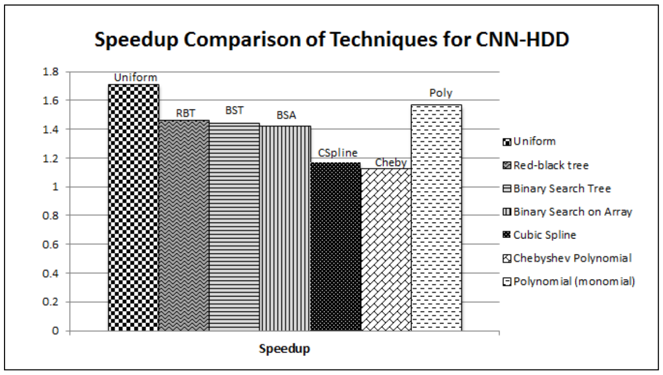

Figure 3. Speedup comparison of the best-case of history-based schemes with that of numerical analysis techniques for the CNN-HDD application.

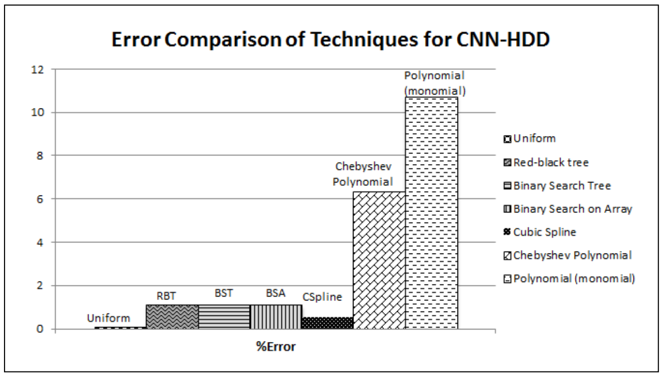

Figure 4. Error comparison of the best-case of history-based schemes with that of numerical analysis techniques for the CNN-HDD application.

we will extend the techniques to support multivariate problems. In the presented work, we manually added the approximation code in the application source code. We will develop compiler techniques to perform these tasks automatically. To set different parameters of the approximation scheme, we also plan to develop automatic tuning techniques.

V. Related Work

Approximating functions has been a major topic in the field of Numerical Analysis. Different techniques like series expansion, interpolation, polynomial approximation [7], and piecewise approximation (splines) aim at approximating functions [8]. Among these, Chebyshev polynomials and cubic splines are generally considered the best approximation schemes. Although our scheme falls in the category of piecewise approximation, it is different from other techniques in many ways. The major difference is the learning ability of our scheme. The other schemes statically replace the function with a polynomial (or polynomials) whereas our scheme learns to approximate. Among other differences are that: other schemes do not keep history, which our scheme consults as needed; other schemes do not have a strict criterion regarding how regions are formed, we form regions based on the history; and other approaches generally use fewer regions and high-order polynomials, whereas we use many regions and lower-order polynomials.

Machine learning also offers different techniques that can be used to approximate functions. Among these are artificial neural networks (ANNs), support vector machines (SVMs) and fitness approximations [9] used in Genetic Algorithms, to name a few. These techniques, however, are only useful for functions that are computationally intensive such as simulations or fitness functions in evolutionary algorithms for complex real-world applications.

The approximate computing community has explored approximation at the levels of hardware [10], architecture [11], data types [12], instructions, loops [13], and synchronization blocks [14], but not so much at the level of procedures. The two attempts that can be said to be applicable to function approximation are a neural network [1] and an approximate memoization technique [2]. The first paper shows that the neural network approach towards approximation for general applications can be effective only if implemented in hardware, as the software implementations degrade performance. This result is further confirmed by [15]. The second paper [2] only focuses on GPU applications and describes approximate memoization as a pattern-based approximation technique for data-parallel applications that is applicable to map patterns found in such applications. This approximation scheme also differs from ours as it uses table lookup to conduct approximate memoization; our scheme uses history-based piecewise approximation with polynomials.

VI. Conclusion

While the field of approximate computing has seen significant growth in recent years, many opportunities remain. Approximating procedures for speed is one such opportunity, which we have explored in this paper. We have introduced a history-based piecewise approximation scheme with two variants, one that forms non-uniform regions and the other that forms uniform regions. We have also described four different realizations of these schemes. Our experiments with 90 functions from GSL show an average speedup of 9.3x for 71% of functions with a small normalized RMS error (0.06) in approximation, and 9.5x for another 15% of functions with a normalized RMS error of 0.49. Our results on applications demonstrate that it is practical to apply the proposed schemes on real applications with substantial performance benefits. We have presented results on three applications, one of which, CNN-HDD represents the class of CNN and deep learning neural network applications. We expect our results to hold for the class of applications. The average speedup of three applications using our scheme is 1.74x with an average error of 0.5%.

References

[1] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[2] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 35–50.

[3] GNU, "GSL," http://www.gnu.org/software/gsl/.

[4] B. Gough, *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms 2nd edition," 2001.

[6] C. Bienia and K. Li, *Benchmarking modern multiprocessors*. Princeton University USA, 2011.

[7] R. Bellman, R. Kalaba, and B. Kotkin, "Polynomial approximation–a new computational technique in dynamic programming: Allocation processes," *Mathematics of Computation*, vol. 17, no. 82, pp. 155–161, 1963.

[8] P. J. Davis, *Interpolation and approximation*. Courier Corporation, 1975.

[9] Y. Jin and B. Sendhoff, "Fitness approximation in evolutionary computation-a survey," in *Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers Inc., 2002, pp. 1105–1112.

[10] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: systematic logic synthesis of approximate circuits," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 796–801.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 301–312.

[12] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.

[13] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[14] M. C. Rinard, "Unsynchronized techniques for approximate parallel computing," in *RACES Workshop*, 2012.

[15] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "Benchnn: On the broad potential application scope of hardware neural network accelerators," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 36–45.