Neural Network-Based Accelerators for Transcendental Function Approximation

Schuyler Eldridge*, Florian Raudies†, David Zou*, and Ajay Joshi*

*Department of Electrical and Computer Engineering, Boston University
†Center for Computational Neuroscience and Neural Technology, Boston University
{schuye, fraudies, f2rf2r, joshi}@bu.edu

ABSTRACT

The general-purpose approximate nature of neural network (NN) based accelerators has the potential to sustain the historic energy and performance improvements of computing systems. We propose the use of NN-based accelerators to approximate mathematical functions in the GNU C Library (`glibc`) that commonly occur in application benchmarks. Using our NN-based approach to approximate cos, exp, log, pow, and sin we achieve an average energy-delay product (EDP) that is 68x lower than that of traditional `glibc` execution. In applications, our NN-based approach has an EDP 78% of that of traditional execution at the cost of an average mean squared error (MSE) of 1.56.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*adaptable architectures, neural nets*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*algorithms implemented in hardware*

General Terms

Design, Measurement, Performance

Keywords

Bio-inspired Computing; Neuromorphic Architectures

1. INTRODUCTION

As transistors scale into the nanometer regime, sustaining the scale of energy and performance improvements that were earlier possible with every new technology generation is becoming increasingly difficult [4, 10]. Dennard Scaling-based improvements [8] are approaching their limits and continued improvements are expected to come from "More-than-Moore" architectural improvements [1]. Hence, there is a critical need to develop novel architectures and programming models that can sustain the historic trends of energy and performance improvements in computing systems.

GLSVLSI'14, May 21–23, 2014, Houston, Texas, USA.

http://dx.doi.org/10.1145/2591513.2591534.

Computational accelerators offer one method of decreasing energy and improving performance. Accelerators can be used alongside traditional processor datapaths to speed up the execution of operations. In such an architecture the processor uses either the traditional datapath or the accelerator during execution. However, most accelerators are special purpose and cannot be leveraged by all applications which limits their use. A truly general-purpose computational accelerator that complements and accelerates the traditional Von Neumann architecture broadly would go a long way towards improving the energy and performance of future computing systems.

One class of accelerators that has the potential to provide general-purpose acceleration are NNs. NNs can be configured to execute a variety of operations to arbitrary accuracy which makes them suitable for general-purpose computation. Unfortunately, the approximate nature of NN-based accelerators limits their potential application. However, the inherent unreliability of current and future nanoscale CMOS devices has prompted researchers to develop a new computational paradigm based on approximate execution of operations. NN-based accelerators, with their approximate nature, can therefore be a good fit for systems based on an approximate computational paradigm.

Bit-level approximation of instructions or functions is an alternative approach to improving performance and reducing energy consumption in current and future CMOS devices. This is particularly appealing when the underlying function can be approximated by simply reducing the number of bits used in the function (e.g., bit truncation of instructions). The large energy requirements of high precision instructions can be significantly reduced if only the necessary precision for an application is used [21]. However, this type of approximation does not reduce the total number of operations that a processor executes. Consequently, large performance benefits and high output accuracies are difficult to achieve.

GPUs and CPUs have dedicated hardware for accelerating certain transcendental functions. However, the large power and data transfer overheads of GPUs make them impractical unless an application has been explicitly written to exploit a SIMD architecture. Additionally, the general-purpose approximate nature of NN-based accelerators has the benefit of allowing one NN-based accelerator to approximate different functions without architectural modifications as would be required of GPUs and CPUs.

In this paper, we present a reconfigurable NN-based accelerator architecture that approximates floating point tran-

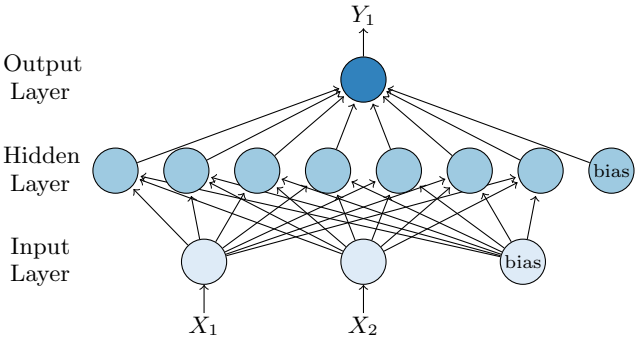

Figure 1: Example two-layer MLP NN.

scendental functions to decrease the EDP of these functions and modern applications. The choice of transcendental functions is driven by two observations. First, floating point instructions show general resilience to approximation [9]. Second, we have found that certain modern recognition, mining, and synthesis (RMS) applications in the PARSEC suite [2] have appreciable numbers of transcendental function calls. Additionally, RMS applications are known to exhibit resilience to approximation [6]. The specific contributions of this paper are as follows:

- We design and implement an accelerator architecture using multilayer perceptron (MLP) NNs in the NCSU FreePDK 45nm predictive technology model [19]. We then compare the execution of transcendental functions using our NN-based accelerator to traditional execution with `glibc`. We compare these implementations using EDP and accuracy as metrics.
- Our proposed accelerator architecture provides close to 2 orders of magnitude improvement in EDP at the cost of an average MSE of 9.04×10^{-3} across five different transcendental functions.
- In addition, we evaluate the EDP benefits and accuracy trade-offs for benchmarks in the PARSEC suite when using our NN-based accelerator to execute transcendental functions.

2. RELATED WORK

The general-purpose approximate nature of NNs has been shown by Cybenko and Hornik [7, 14]. The acceleration of general-purpose applications (e.g. filtering, FFT) by approximating regions of code with MLP networks has been proposed and shown to provide power and performance benefits if the granularity of approximation is sufficiently large and the approximated code is executed frequently [11]. Furthermore, different types of NNs have been used to approximate large regions of code or whole programs in applications [5, 15]. Additional approaches explore alternative benefits that NN-based systems may provide such as fault-tolerance [12, 13, 20].

Our work extends these prior approaches by utilizing NN-based accelerators at the level of library functions as opposed to larger functions or applications. Additionally, our NN-based accelerators are able to run without static/run-time code analysis or application failure.

Table 1: Expected, median, and minimum MSE for 100 2-layer, 7 hidden node, 9 fractional bit MLP NNs trained to execute transcendental functions.

Func.	E[MSE]	M[MSE]	Min[MSE]	Domain
sin	5.3×10^{-4}	4.3×10^{-4}	0.4×10^{-4}	$[0, \frac{\pi}{4}]$
cos	5.6×10^{-4}	4.1×10^{-4}	0.2×10^{-4}	$[0, \frac{\pi}{4}]$
asin	21.7×10^{-4}	18.5×10^{-4}	4.9×10^{-4}	$[-1, 1]$
acos	19.0×10^{-4}	16.5×10^{-4}	5.5×10^{-4}	$[-1, 1]$
exp	6.3×10^{-4}	4.4×10^{-4}	1.0×10^{-4}	$[-\log 2, \log 2]$
log	12.1×10^{-4}	8.4×10^{-4}	0.4×10^{-4}	$[\frac{1}{2}, 1)$

3. NEURAL NETS AS APPROXIMATORS

3.1 Theory

MLP NNs are layered structures that process data in a feedforward manner. The input layer of the NN receives input data, the data is processed by hidden neurons in one or more hidden layers, and the output layer of the NN produces one or more outputs. Figure 1 shows a 2-layer NN[1] with one input layer, one hidden layer, and one output layer. The output, y, of a neuron in the hidden or output layer is determined by an activation function ϕ evaluated using the sum of all N input edges $x_0, x_1, \ldots x_{N-1}$ multiplied by each corresponding edge weight $w_0, w_1, \ldots w_{N-1}$:

$$y = \phi\left(\sum_{k=0}^{N-1} w_k x_k\right) \tag{1}$$

While the activation function can take many forms, we find that sigmoid hidden units and linear output units are able to accurately approximate the transcendental functions that we analyze:

$$\text{sigmoid:} \qquad \phi = \frac{1}{1 + e^{-2s_1 x}} \tag{2}$$

$$\text{linear:} \qquad \phi = s_2 x \tag{3}$$

The steepness parameters that govern the sharpness of the sigmoid, s_1, and the slope of the linear units, s_2, are both set to one. Input neurons are pass-through and do not modify their inputs.

An NN can be trained to approximate a function through error backpropagation that adjusts weights to minimize the output error using a Euclidean distance metric (i.e., gradient descent) [18]. Other algorithms use different distance metrics or approaches to accelerate convergence, i.e., Resilient Backpropagation [17] used by the Fast Artificial Neural Network (FANN) library [16]. The universal approximation ability of NNs can be exploited to enable one NN to approximate different functions. We illustrate this by showing the ability of one NN to approximate several different transcendental functions. We train 100 randomly initialized NNs (1 input neuron, 7 hidden neurons, and 1 output neuron) to separately compute transcendental functions over limited input domains using training and validation datasets. The expected, median, and minimum MSE of the validation datasets are shown in Table 1. The choice of 7 hidden neurons is arbitrary and is only meant to qualitatively validate the ability of NNs to act as generic approximators. Note that the chosen, limited input domains decrease the complexity

[1]We use the general convention of counting NN layers as the number of hidden layers plus one for the output layer.

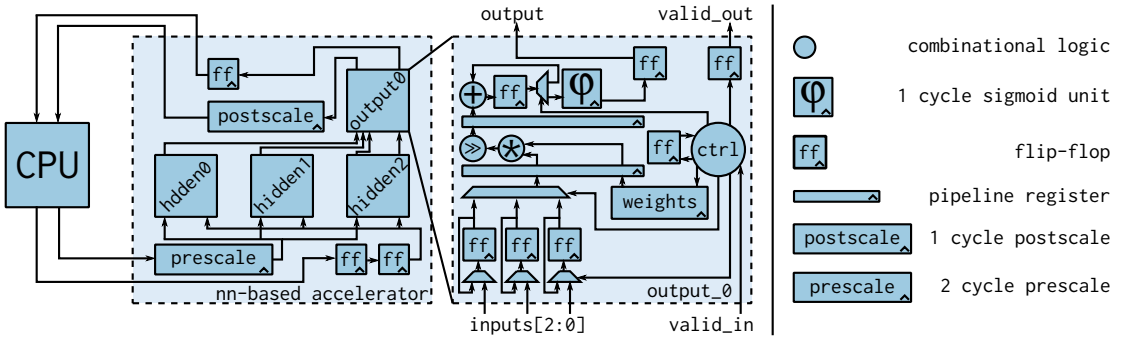

Figure 2: Block diagram of an NN-based accelerator with 3 hidden neurons and 1 output neuron. Input neurons are pass-through and not shown. The internals of an output neuron are shown in the middle and a legend on the right. Each neuron uses a single multiplier, a single accumulator, and a piecewise linear approximation unit. Processing is pipelined within a neuron as well as across layers.

Table 2: Identities from Walther [23] to convert full-domain inputs onto finite domains d for the CORDIC algorithm [7].

Identity	Domain
$\sin(d+\frac{q\pi}{2}) = \begin{cases} \sin(d) & \text{if } q\%4=0 \\ \cos(d) & \text{if } q\%4=1 \\ -\sin(d) & \text{if } q\%4=2 \\ -\cos(d) & \text{if } q\%4=3 \end{cases}$	$0 < d < \frac{\pi}{2}$
$\cos(d+\frac{q\pi}{2}) = \begin{cases} \cos(d) & \text{if } q\%4=0 \\ -\sin(d) & \text{if } q\%4=1 \\ -\cos(d) & \text{if } q\%4=2 \\ \sin(d) & \text{if } q\%4=3 \end{cases}$	$0 < d < \frac{\pi}{2}$
$\log(d2^q) = \log(d) + q\log(2)$	$\frac{1}{2} \le d < 1$
$\exp(q\log 2 + d) = 2^q \exp(d)$	$\lvert d\rvert < \log 2$

Table 3: Scaling steps using Table 2 identities.

Step	$\sin x, \cos x$	$\exp x$	$\log x$
$f(d,q)$	$x = \frac{q\pi}{2} + d$	$x = q\log 2 + d$	$x = d2^q$
pre-1	$q = \lfloor \frac{2x}{\pi} \rfloor$	$q = \lfloor \frac{x}{\log 2} + 1 \rfloor$	$q = \lceil \lg x \rceil$
pre-2	$d = x - \frac{q\pi}{2}$	$d = x - q\log 2$	$d = x >> q$
NN	$y = \text{NN}(d)$	$d = \text{NN}(d)$	$d = \text{NN}(d)$
post	not needed	$y = d << q$	$y = d + q\log 2$

of the input-output surface that the NNs need approximate. Training an NN to directly approximate one of these functions, e.g., $\sin x$, on an unbounded domain is intractable. So long as the input-output surface of the NN is representative of the whole function, input prescaling and output postscaling methods can be developed that allow an NN, that computes over a limited domain, to effectively compute over the full input domain of the whole function.

For our NN-based approximator, we implement prescaling and postscaling steps using mathematical identities and scalings of the CORDIC algorithm [22, 23] and shown in Tables 2 and 3. For example, say we want to compute $\sin x$ for any x using our NN, but our NN cannot handle unbounded inputs. We train our NN to compute $\sin d$ and $\cos d$ for all d on the limited domain $(0, \frac{\pi}{2})$. Knowing that we can compute $\sin x$ as $\sin(d+\frac{q\pi}{2})$ we find d and q and compute the answer for any x using our limited domain NN.

The two prescaling steps in Table 3 are used to find q and d using computationally easy operations, i.e., multiplication with a constant, addition/subtraction, and bit manipulations. Post scaling operations are necessary for exp and log, but not sin and cos. The pow function cannot be computed using an identity. However, it can be computed as a combination of log and exp as follows:

$$a^b = e^{b\log a} \tag{4}$$

We use these identities and scalings to implement an NN-based accelerator for transcendental functions.

3.2 Implementation

In this section, we describe the architecture of our proposed NN-based accelerator. The architecture for log and exp is shown in Figure 2. The architecture is composed of prescaling and postscaling units, hidden neurons (`hidden0`, `hidden1`, and `hidden2`) and an output neuron (`output0`). The input layer is pass-through and not shown. Processing is pipelined across neurons, i.e., the hidden layer can operate on data while the output layer is operating on the previous data. Additionally, weight–input multiplication and accumulation is processed in a 3-stage, internal neuron pipeline. The accelerator works alongside a CPU and operates as follows.

When the CPU encounters one of the transcendental functions supported by our NN-based accelerator, the input values are passed to the accelerator by means of multiplexed connections. The NN-based accelerator thereby acts as an additional execution unit. We consequently assume this connection is implemented with combinational logic and does add to the latency and energy of the architecture. Data arriving at the NN-based accelerator passes through a 2-cycle prescaling stage (see Table 3) before entering the network along with a data valid flag. Each hidden neuron then operates in parallel on its inputs. A neuron latches input data and begins computation when it sees a data valid signal (`valid_in`). A neuron then multiplies each input by its corresponding weight in a 3-stage pipeline. The input from the bias neuron is implemented as the starting value for the accumulation flip-flop. The accumulated sum is passed through a 1-cycle activation function, ϕ. Each hidden neuron output is then sent to the output unit while a data valid signal (`valid_out`) is asserted. The output unit processes data similarly and returns data to the CPU through a postscaling unit (see Table 3). In the case of sin and cos, postscaling is not necessary and data is returned directly to the CPU. Processing is pipelined across NN layers and within a neuron.

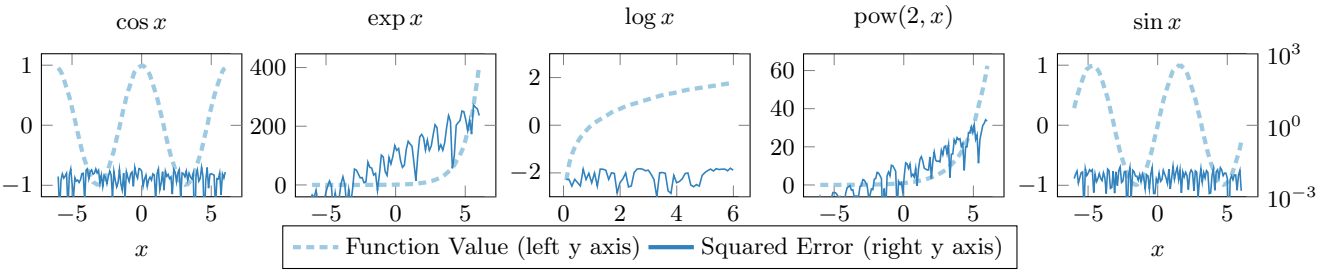

Figure 3: NN-based functions and their errors. Note: Error is plotted on a log scale using the right y axis.

The latency and throughput for a single neuron can be defined as follows, where N is the number inputs to a neuron:

$$\begin{aligned} \text{latency} &= 6 + N - 1 = N + 5 \\ \text{throughput} &= \frac{1}{N+5} \end{aligned} \tag{5}$$

The latency of the entire NN is a function of the number of inputs to the NN (N_i), the number of neurons in the hidden layer (N_h, i.e., the number of inputs to the output neuron), and the latency of the scaling stages (L_s). The throughput is the inverse latency of the longest stage (i.e., either the hidden layer or the output layer). The scaling latency is two for cos and sin and three for log and exp.

$$\begin{aligned} \text{latency} &= L_s + (N_i + 5) + (N_h + 5) \\ \text{throughput} &= \frac{1}{\max(N_i, N_h) + 5} \end{aligned} \tag{6}$$

4. EVALUATION

In this section, we compare accelerator-based execution with traditional execution of transcendental functions using EDP and accuracy metrics. We also explore the impact of NN-based accelerator usage on overall application behavior.

4.1 Accelerator-Based Execution versus Traditional Execution

To compare the accelerator-based execution and the traditional execution of transcendental functions, we use the Verilog hardware description language (HDL) to implement 2-layer NN designs in hardware with 1–15 hidden neurons and 6–10 fractional bits. We select these ranges because we find that NNs with too many hidden neurons are prone to overtraining and NNs with fewer than 6 fractional bits show high error rates. NNs are then synthesized and placed-and-routed (PnR) using a Cadence toolflow that uses the NCSU FreePDK 45nm predictive technology model to generate the final hardware design. All designs are run at their maximal possible frequencies as determined by the Cadence tools. We determine the energy using PnR tools with random data applied to input neurons. We evaluate the accuracy of these configurations by testing 100 trained instances of each NN configuration for each transcendental function. We use the FANN library to implement train and test these NNs in software.

For each transcendental function we select the NN configuration that minimizes the Energy-Delay-Error Product (EDEP) metric, which we define as follows:

$$\text{EDEP} = \text{energy} \times \frac{\text{latency in cycles}}{\text{frequency}} \times \text{MSE} \tag{7}$$

Table 4: NN-based accelerator hardware parameters with minimum EDEP for sin, cos, log, and exp.

NN	Func.	Area (um^2)	Freq. (MHz)	Energy (pJ)
`h1_b6`	cos,sin	1259.50	337.38	8.30
`h3_b7`	exp,log	3578.50	335.80	24.81

Table 5: MSE and energy consumption of our NN-based implementation of transcendental functions.

Func.	NN	MSE	Energy (pJ)
cos	`h1_b6`	9.38×10^{-4}	8.30
exp	`h3_b7`	1.68×10^{-4}	24.81
log	`h3_b7`	1.45×10^{-4}	24.81
pow	`h3_b7`	4.32×10^{-2}	101.67
sin	`h1_b6`	7.37×10^{-4}	8.30

The parameters of the networks with a minimum EDEP for sin, cos, log, and exp are shown in Table 4. We find that sin and cos have minimal EDEP for a network with 1 hidden neuron and 6 fractional bits (abbreviated `h1_b6`). A network with 3 hidden neurons and 7 fractional bits (`h3_b7`) has minimal EDEP for log and exp. Using prescaling and postscaling, the outputs and squared error of these networks are shown in Figure 3. Error is plotted on a log scale on the right axis. We additionally show the output and error for pow which is computed using a combination of log and exp. Figure 3 shows that our NN-based accelerator can be used to approximate these five transcendental functions. Additionally, error and output scale together, i.e., a small output has a small error and a large output has a larger error. Table 5 shows the NN configuration, MSE and energy consumed by the NN-based execution of the transcendental functions.

Traditional execution of the transcendental functions involves executing a series of floating point instructions including addition, subtraction, and multiplication. Table 6 shows the average number of instructions that are executed for each transcendental function. We generate this table using the gem5 simulator [3] to trace small programs that repeatedly call transcendental functions with random inputs. We process these traces to pull out only those instructions related to transcendental function calls. Control, integer, and move instructions are aggregated in separate columns. Instructions executed less frequently than floating point subtraction are not shown in the table. Due to random inputs, the average number of instructions per function is not an integer. This is expected because `glibc` executes different code paths depending on input value.

For a valid EDP comparison of the NN-based execution with traditional `glibc` execution, we compute the energy per

Table 6: Mean floating point instruction counts (ss denotes single precision and sd denotes double precision) in single precision (e.g., cosf) and double precision (e.g., cos) glibc transcendental functions. Fractional instruction counts occur because glibc takes different code paths based on the random input values used. The estimated energy per function is shown using energy per instruction data from Table 7.

Func.	addsd	addss	mulsd	mulss	subsd	subss	Total Instructions	Energy (pJ)
cos	7.42	0.00	11.64	0.00	8.41	0.00	114.55	966.61
cosf	0.00	3.00	0.00	10.00	0.00	7.03	103.45	365.13
exp	11.00	0.00	14.00	0.00	6.00	0.00	159.97	1158.10
expf	5.00	1.00	5.00	1.00	2.00	1.00	218.00	453.16
log	18.06	0.00	11.79	0.00	5.09	0.00	226.86	994.72
logf	0.00	7.69	0.00	11.38	0.00	3.65	143.09	415.44
pow	32.46	0.00	30.54	0.00	20.54	0.00	337.53	2561.33
powf	0.00	23.32	0.00	35.00	0.00	26.32	354.56	1292.49
sin	8.17	0.00	10.97	0.00	5.99	0.00	109.40	909.30
sinf	0.00	3.00	0.00	8.54	0.00	5.16	96.68	311.42

Table 7: Parameters of traditional glibc implementations of floating point instructions.

Instruction	Area (um^2)	Freq. (MHz)	Energy (pJ)
addss	635.5	388	1.00
addsd	1466.7	388	2.20
mulss	6505.3	283	35.51
mulsd	16226.5	135	80.05

Table 8: EDP of NN-based and traditional glibc execution of transcendental functions.

Func.	EDP-NN	EDP-Single	EDP-Double
cos	3.44×10^{-19}	1.89×10^{-17}	5.54×10^{-17}
exp	1.26×10^{-18}	3.62×10^{-16}	9.26×10^{-17}
log	1.26×10^{-18}	2.97×10^{-17}	1.13×10^{-16}
pow	1.05×10^{-17}	2.29×10^{-16}	4.32×10^{-16}
sin	3.44×10^{-19}	1.51×10^{-17}	4.97×10^{-17}

floating point instruction using the same Cadence toolflow. The area, maximum operating frequency, and energy for floating point addition and multiplication is shown in Table 7. Floating point subtraction is taken to be equivalent to floating point addition. Other instructions are not evaluated as we find that floating point addition, multiplication, and subtraction are the most frequently executed instructions (after move instructions) in glibc transcendental functions.

Using the floating point addition, multiplication, and subtraction instruction counts and energy per instruction in Table 7, we calculate the energy per traditional transcendental function. Energy per transcendental function is shown in the last column of Table 6. We then compare the EDP of traditional and NN-based implementations. We did not use EDEP as the comparison metric because the MSE of a glibc implementation is effectively zero. We assume that instructions in transcendental functions can achieve an IPC of one at 2 GHz. The EDP comparison against single and double precision glibc implementations is shown in Table 8. The EDP value for our NN-based execution is, averaging between single and double precision, 68x lower than that of traditional execution. Our NN-based design trades off an average transcendental function MSE of 9.04×10^{-3} for this EDP savings.

Table 9: Percentage of total application cycles spent in transcendental functions and the estimated EDP of an NN-based accelerator implementation normalized to traditional, single precision floating point execution. Applications in the lower division have no transcendental functions.

Benchmark	% Total Cycles	Normalized EDP
blackscholes	45.65%	0.5583
bodytrack	2.25%	0.9783
canneal	1.19%	0.9885
swaptions	39.33%	0.6191
dedup	0.00%	1.0000
fluidanimate	0.00%	1.0000
freqmine	0.00%	1.0000
raytrace	0.00%	1.0000
streamcluster	0.00%	1.0000
x264	0.00%	1.0000
vips	0.00%	1.0000

4.2 NN-Based Accelerators in Applications

We analyze the EDP versus accuracy trade-offs associated with using our NN-based accelerators in applications. We execute benchmarks in the PARSEC suite using the gem5 simulator and record cycle counts. We then determine the percentage of total cycles that benchmarks spend executing transcendental functions. The EDP savings for each benchmark then follow an Amdahl's law convention—total EDP savings are limited by the number of cycles each benchmark spends executing transcendental functions. Table 9 shows the normalized EDP savings by applying the single precision floating point EDP reductions in Table 8. Results for facesim and ferret are excluded because cycle counts were not able to be obtained using gem5. Some applications do not use transcendental functions. However, NN-based accelerators may be reconfigured to approximate small or large portions of these applications for EDP savings.

We then build a software library that redefines the execution of selected transcendental functions (cos, exp, log, pow, and sin). Our software implementation uses the NN configurations listed in Table 5 for executing each transcendental function. We execute benchmarks in the PARSEC suite and compare their outputs to those of traditional execution of PARSEC benchmarks that rely on traditional glibc. For

Table 10: Application output MSE and percent error using NN-based accelerators.

Benchmark	MSE	E[|%error|]
`blackscholes`	4.48×10^{-1}	24.6236%
`bodytrack`	2.07×10^{-1}	29.6321%
`canneal`	2.89×10^{8}	0.0025%
`ferret`	1.03×10^{-3}	1.9633%
`swaptions`	5.59×10^{0}	36.8205%

this analysis, we only execute benchmarks that have transcendental function calls: `blackscholes`, `bodytrack`, `canneal`, `ferret`, and `swaptions`. We report the MSE and expected percent absolute error of application outputs in Table 10. The MSE and expected percent absolute error for all benchmarks is in an acceptable range. The output for `canneal` is a single large value, hence the large MSE, but low percent error.

Overall, NN-based accelerators approximating transcendental functions are able to provide substantial EDP reductions over standard `glibc` implementations. However, EDP and performance gains (and accuracy trade-offs) are explicitly governed by an Amdahl's law convention—any gains or losses that NN-based accelerators provide are directly proportional to the percentage of time that an application spends using the approximated functions. Benchmarks `blackscholes` and `swaptions` spend approximately 40% of their execution time computing floating point transcendental functions and can consequently decrease their EDP substantially. Benchmarks `dedup`, `freqmine`, `raytrace`, `streamcluster`, and `x264` do not use any of the transcendental functions that we approximate. These benchmarks consequently see no EDP improvements or accuracy reductions. Benchmarks `bodytrack` and `canneal` make limited use of approximated transcendental functions and see modest EDP gains and small accuracy losses. All applications executed ran to completion without runtime errors.

5. CONCLUSION

NN-based accelerators provide a malleable substrate on which different types of approximate computations can be executed. Our work demonstrates that their inclusion in future architectural designs has the potential to reduce EDP at the cost of application output error. A hardware-level comparison of NN-based and `glibc` execution of transcendental functions shows that the EDP for NN-based execution is, on average, 68x lower than that of `glibc` execution. Moreover, the use of an NN-based accelerator to approximate transcendental functions in PARSEC benchmarks uses, averaged across the 5 benchmarks with transcendental functions, 78% of the EDP of a traditional `glibc` implementation. Output MSE and percent absolute error are, on average, 1.56 and 0.24, excluding `canneal`. These results indicate that library-level approximation may be a viable direction for leveraging energy reductions while maintaining safe program execution. This work can be furthered by using NN-based accelerators to approximate additional functions in `glibc` and other libraries.

6. ACKNOWLEDGMENTS

This work was supported by a NASA Office of the Chief Technologist's Space Technology Research Fellowship.

7. REFERENCES

[1] W. Arden, et al. More-than-moore. *International Technology Roadmap for Semiconductors*, 2010.

[2] C. Bienia et al. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[3] N. Binkert, et al. The gem5 simulator. *ACM Comp. Ar.*, 39(2):1–7, 2011.

[4] S. Borkar et al. The future of microprocessors. *Comm. ACM*, 54(5):67–77, 2011.

[5] T. Chen, et al. Benchnn: On the broad potential application scope of hardware neural network accelerators. In *IISWC*, 2012.

[6] V. K. Chippa, et al. Analysis and characterization of inherent application resilience for approximate computing. In *DAC*, 2013.

[7] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control Signal*, 2(4):303–314, 1989.

[8] R. H. Dennard, et al. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE J. Solid-St. Circ.*, 9(5):256–268, 1974.

[9] H. Duwe. Exploiting application level error resilience via deferred execution. *Master's thesis, University of Illinois at Urbana Champaign*, 2013.

[10] H. Esmaeilzadeh, et al. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[11] H. Esmaeilzadeh, et al. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[12] A. Hashmi, et al. Automatic abstraction and fault tolerance in cortical microachitectures. In *ISCA*, 2011.

[13] A. Hashmi, et al. A case for neuromorphic isas. *ACM SIGPLAN Notices*, pages 145–158, 2011.

[14] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[15] B. Li, et al. Memristor-based approximated computation. In *ISLPED*, 2013.

[16] S. Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Department of Computer Science University of Copenhagen (DIKU), 2003. http://fann.sf.net.

[17] M. Riedmiller et al. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *ICNN*, 1993.

[18] D. E. Rumelhart et al. *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations*. MIT Press, Cambridge, Ma, 1986.

[19] J. E. Stine, et al. Freepdk: An open-source variation-aware design kit. In *MSE*, 2007.

[20] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *ISCA*, 2012.

[21] S. Venkataramani, et al. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.

[22] J. E. Volder. The cordic trigonometric computing technique. *IRE Tran. Comput.*, EC-8(3):330 –334, sept. 1959.

[23] S. Walther. A unified algorithm for elementary functions. *AFIPS*, 1971.