Toward General-Purpose Code Acceleration with Analog Computation

Amir Yazdanbakhsh[*] Renee St. Amant[§] Bradley Thwaites[*] Jongse Park[*]
Hadi Esmaeilzadeh[*] Arjang Hassibi[§] Luis Ceze[†] Doug Burger[‡]
[*]Georgia Institute of Technology [§]The University of Texas at Austin [†]University of Washington [‡]Microsoft Research

Abstract

We propose a solution—from circuit to compiler—that enables general-purpose use of limited-precision, analog hardware to accelerate "approximable" code—code that can tolerate imprecise execution. We utilize an algorithmic transformation that automatically converts approximable regions of code from a von Neumann model to an "analog" neural model. We outline the challenges of taking an analog approach, including restricted-range value encoding, limited precision in computation, circuit inaccuracies, noise, and constraints on supported topologies. We address these limitations with a combination of circuit techniques, a novel hardware/software interface, neural-network training techniques, and compiler support. Analog neural acceleration provides whole application speedup of 3.3× and and energy savings of 12.1× with quality loss less than 10% for all except one benchmark. These results show that using limited-precision analog circuits for code acceleration, through a neural approach, is both feasible and beneficial over a range emerging applications.

1. Introduction

A growing body of work [11, 25, 6, 22, 12] has focused on *approximation* as a strategy for improving performance and efficiency through approximation. Large classes of applications can tolerate small errors in their outputs with no discernible loss in *Quality of Result* (QoR). Many conventional techniques in energy-efficient computing navigate a design space defined by the two dimensions of performance and energy, and traditionally trade one for the other. General-purpose approximate computing explores a third dimension—that of error.

Many interesting design alternatives become possible once precision is relaxed. An obvious candidate is the use of analog circuits for computation. However, computation in the analog domain has several major challenges, even when small errors are permissible. First, analog circuits tend to be special purpose, good for only specific operations. Second, the bit widths they can accommodate are much smaller than current floating-point standards (i.e. 32 or 64 bits), since the ranges must be represented by physical voltage or current levels. Another consideration is determining where the boundaries between digital and analog computation lie. Using individual analog operations will not be effective due to the overhead of A/D and D/A conversions. Finally, effective storage of temporary analog results is challenging in current CMOS technologies. Due to these limitations, it has not been effective to design analog von Neumann processors that can be programmed with conventional languages. NPUs can be a potential solution for general-purpose analog computing. Prior research has shown that analog neural networks can effectively solve classes of domain-specific problems, such as pattern recognition [4, 27, 28, 18]. The process of invoking a neural network and returning a result defines a clean, coarse-grained interface for D/A and A/D conversion. Furthermore, the compile-time training of the network permits any analog-specific restrictions to be hidden from the programmer. The programmer simply specifies which region of the code can be approximated, without adding any neural-network-specific information. Thus, no additional changes to the programming model are necessary. Figure 1 illustrates an overview of our framework.

This paper reports on our study to design an NPU with mixed-signal components and develop a compilation workflow for utilizing the mixed-signal NPU for code acceleration. The goal of this study is to investigate challenges and potential solutions of implementing NPUs with analog components, while both bounding application error to sufficiently low levels and achieving worthwhile performance or efficiency gains for general-purpose approximable code. We found that exposing the analog limitations to the compiler allowed for the compensation of these shortcomings and produced sufficiently accurate results. We trained networks at compile time using 8-bit values, topologies restricted to eight inputs per neuron, plus RPROP and CDLM [8] for training. Using these techniques together, we were able to bound error on all applications to a 10% limit, which is comparable to prior studies using entirely digital accelerators. The average time required to compute a neural result was 3.3× better than a previous digital implementation with an additional energy savings of 12.1×. The performance gains result in an average full-application-level improvement of 3.7× and 23.3× in performance and energy-delay product, respectively. This study shows that using limited-precision analog circuits for code acceleration, by converting regions of imperative code to neural networks and exposing the circuit limitations to the compiler, is both feasible and advantageous.

2. Analog Circuits for Neural Computation

As Figure 2a illustrates, each neuron in a multi-layer perceptron takes in a set of inputs (x_i) and performs a weighted sum of those input values ($\sum_i x_i w_i$). The weights (w_i) are the result of training the neural network on . After the summation stage, which produces a linear combination of the weighted inputs, the neuron applies a nonlinearity function, *sigmoid*, to the result of summation. Figure 2b depicts a conceptual analog circuit that performs the three necessary operations of a neuron: (1) scaling inputs by weight ($x_i w_i$), (2) summing the scaled inputs ($\sum_i x_i w_i$), and (3) applying the nonlinear-

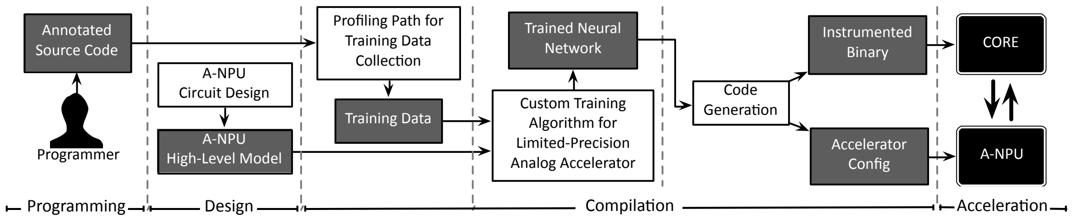
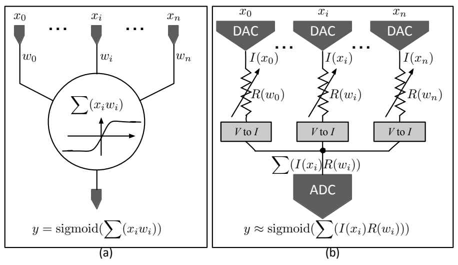


Figure 1: Framework for using limited-precision analog computation to accelerate code written in conventional languages.



x_0 x_i x_n
w_0 w_i w_n
$\sum(x_i w_i)$
$y = \text{sigmoid}(\sum(x_i w_i))$
(a)
DAC
$I(x_0)$ $I(x_i)$ $I(x_n)$
$R(w_0)$ $R(w_i)$ $R(w_n)$
V to I
$\sum(I(x_i)R(w_i))$
ADC
$y \approx \text{sigmoid}(\sum(I(x_i)R(w_i)))$
(b)

Figure 2: One neuron and its conceptual analog circuit.

ity function (*sigmoid*). This conceptual design first encodes the digital inputs (x_i) as analog current levels ($I(x_i)$). Then, these current levels pass through a set of variable resistances whose values ($R(w_i)$) are set proportional to the corresponding weights (w_i). The voltage level at the output of each resistance ($I(x_i)R(w_i)$), is proportional to $x_i w_i$. These voltages are then converted to currents that can be summed quickly according to Kirchhoff's current law (KCL). Analog circuits only operate linearly within a small range of voltage and current levels, outside of which the transistors enter saturation mode with IV characteristics similar in shape to a non-linear sigmoid function. Thus, at the high level, the non-linearity is naturally applied to the result of summation when the final voltage reaches the analog-to-digital converter (ADC). Compared to a digital implementation of a neuron, which requires multipliers, adder trees and sigmoid lookup tables, the analog implementation leverages the physical properties of the circuit elements and is orders of magnitude more efficient. However, it operates in limited ranges and therefore offers limited precision.

Analog-digital boundaries. The conceptual design in Figure 2b draws the analog-digital boundary at the level of an algorithmic neuron. As we will discuss, the analog neural accelerator will be a composition of these analog neural units (ANUs). However, an alternative design, primarily optimizing for efficiency, may lay out the entirety of a neural network with only analog components, limiting the D-to-A and A-to-D conversions to the inputs and outputs of the neural network and not the individual neurons. The overhead of conversions in the ANUs significantly limits the potential efficiency gains of an analog approach toward neural computation. However, there is a tradeoff between efficiency, reconfigurability (generality), and accuracy in analog neural hardware design. Pushing more of the implementation into the analog domain gains efficiency at the expense of flexibility, limiting the scope of supported network topologies and, consequently, limiting potential network accuracy. The NPU approach targets *code approximation*, rather than typical, simpler neural tasks, such as recognition and prediction, and imposes higher accuracy requirements. The main challenge is to mange this tradeoff to achieve acceptable accuracy for code acceleration, while delivering higher performance and efficiency when analog neural circuits are used for *general-purpose code acceleration*.

While a holistically analog neural hardware design with fixed-wire connections between neurons may be efficient, it effectively provides a fixed topology network, limiting the scope of applications that can benefit from the neural accelerator, as the optimal network topology varies with application. Additionally, routing analog signals among neurons and the limited capability of analog circuits for buffering signals negatively impacts accuracy and makes the circuit susceptible to noise. In order to provide additional flexibility, we set the digital-analog boundary in conjunction with an algorithmic, sigmoid-activated neuron. where a set of digital inputs and weights are converted to the analog domain for efficient computation, producing a digital output that can be accurately routed to multiple consumers. We refer to this basic computation unit as an analog neural unit, or ANU. ANUs can be composed, in various physical configurations, along with digital control and storage, to form a reconfigurable mixed-signal NPU, or A-NPU.

Value representation and bit-width limitations. One of the fundamental design choices for an ANU is the bit-width of inputs and weights. Increasing the number of bits results in an exponential increase in the ADC and DAC energy dissipation and can significantly limit the benefits from analog acceleration. Furthermore, due to the fixed range of voltage and current levels, increasing the number of bits translates to quantizing this fixed value range to fine granularities that practical ADCs can not handle. In addition, the fine granularity encoding makes the analog circuit significantly more susceptible to noise, thermal, voltage, current, and process variations. In practice, these non-ideal effects can adversely affect the final accuracy when more bit-width is used for weights and inputs. We design our ANUs such that the granularity of the voltage and current levels used for information encoding is to a large degree robust to variations and noise.

Topology restrictions. Another important design choice is the *number of inputs* in the ANU. Similar to bit-width, increasing the number of ANU inputs translates to encoding a larger

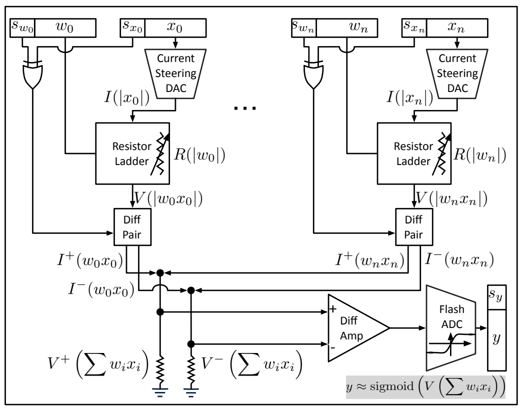


Figure 3: A single analog neuron (ANU).

value range in a bounded voltage and current range, which, as discussed, becomes impractical. The larger the number of inputs, the larger the number of multiply and add operations that can be done in parallel in the analog domain, increasing efficiency. However, due to the bounded range of voltage and currents, increasing the number of inputs requires decreasing the number of bits for inputs and weights. Through circuit-level simulations, we empirically found that limiting the number of inputs to eight with 8-bit inputs and weights strikes a balance between accuracy and efficiency. However, this unique ANU limitation restricts the topology of the neural network that can run on the analog accelerator. Our customized training algorithm and compilation workflow takes into account this topology limitation and produces neural networks that can be computed on our mixed-signal accelerator.

Non-ideal sigmoid. The saturation behavior of the analog circuit that leads to sigmoid-like behavior after the summation stage represents an approximation of the ideal sigmoid. We measure this behavior at the circuit level and expose it to the compiler and the training algorithm.

3. Mixed-Signal Neural Accelerator (A-NPU)

Circuit design for ANUs. Figure 3 illustrates the design of a single analog neuron (ANU). As mentioned, the ANU performs the computation of a single neuron, which is $y \approx sigmoid(\sum_i w_i x_i)$. We place the analog-digital boundary at the ANU level, with computation in the analog domain and storage in the digital domain. Digital input and weight values are represented in sign-magnitude form. In the figure, s_{w_i} and s_{x_i} represent the sign bits and w_i and x_i represent the magnitude. Digital input values are converted to the analog domain through current-steering DACs that translate digital values to analog currents. Current-steering DACs are used for their speed and simplicity. In Figure 3, $I(|x_i|)$ is the analog current that represents the magnitude of the input value, x_i. Digital weight values control resistor string ladders that create a variable resistance depending on the magnitude of each weight ($R(|w_i|)$) . We use a standard resistor ladder thats consists of a set of resistors connected to a tree-structured set of switches. The digital weight bits (w_is) control the switches, adjusting the effective resistance, $R(|w_i|)$, seen by the input current ($I(|x_i|)$). These variable resistances scale the input currents by the digital weight values, effectively multiplying each input magnitude by its corresponding weight magnitude. The output of the resistor ladder is a voltage: $V(|w_i x_i|) = I(|x_i|) \times R(|w_i|)$. The resistor network requires 2^m resistors and approximately 2^{m+1} switches, where m is the number of digital weight bits. This resistor ladder design has been shown to work well for $m \leq 10$. Our circuit simulations show that only minimally sized switches are necessary.

$V(|w_i x_i|)$ as well as the XOR of the weight and input sign bits feed to a differential pair that converts voltage values to two differential current ($I^+(w_i x_i)$, $I^-(w_i x_i)$) that capture the sign of the weighted input. These differential currents are proportional to the voltage applied to the differential pair, $V(|w_i x_i|)$. If the voltage difference between the two gates is kept small, the current-voltage relationship is linear, producing $I^+(w_i x_i) = \frac{I_{bias}}{2} + \Delta I$ and $I^-(w_i x_i) = \frac{I_{bias}}{2} - \Delta I$. Resistor ladder values are chosen such that the gate voltage remains in the range that produces linear outputs, and consequently a more accurate final result. Based on the sign of the computation, a switch steers either the current associated with a positive value or the current associated with a negative value to a single wire to be efficiently summed according to Kirchhoff's current law. The alternate current is steered to a second wire, retaining differential operation at later design stages. Differential operation combats environmental noise and increases gain, the later being particularly important for mitigating the impact of analog range challenges at later stages.

Resistors convert the resulting pair of differential currents to voltages, $V^+(\sum_i w_i x_i)$ and $V^-(\sum_i w_i x_i)$, that represent the weighted sum of the inputs to the ANU. These voltages are used as input to an additional amplification stage (implemented as a current-mode differential amplifier with diode-connected load). The goal of this amplification stage is to significantly magnify the input voltage *range of interest* that maps to the linear output region of the desired sigmoid function. Our experiments show that neural networks are sensitive to the steepness of this non-linear function, losing accuracy with shallower, non-linear activation functions. This fact is relevant for an analog implementation because steeper functions increase range pressure in the analog domain, as a small range of interest must be mapped to a much larger output range in accordance with ADC input range requirements for accurate conversion. We magnify this range of interest, choosing circuit parameters that give the required gain, but also allowing for saturation with inputs outside of this range.

The amplified voltage is used as input to an analog-to-digital converter that converts the voltage to a digital value. We chose a flash ADC design (named for its speed), which consists of a set of reference voltages and comparators [1, 17]. The ADC requires 2^n comparators, where n is the number of digital output bits. Flash ADC designs are capable of converting 8 bits

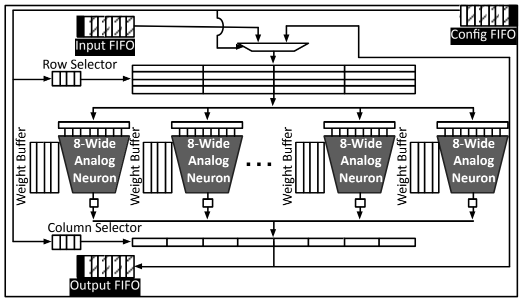


Figure 4: Mixed-signal neural accelerator, A-NPU. Only four of the ANUs are shown. Each ANU processes eight 8-bit inputs.

at frequency on the order of one GHz. We require 2–3 mV between ADC quantization levels for accurate operation and noise tolerance. Typically, ADC reference voltages increase linearly; however, we use a non-linearly increasing set of reference voltages to capture the behavior of a sigmoid function which also improves the accuracy of the analog sigmoid.

Reconfigurable mixed-signal A-NPU. We design a reconfigurable mixed-signal A-NPU that can perform the computation of a wide variety of neural topologies since each requires a different topology. Figure 4 illustrates the A-NPU design with some details omitted for clarity. The figure shows four ANUs while the actual design has eight. The A-NPU is a time-multiplexed architecture where the algorithmic neurons are mapped to the ANUs based on a static scheduling algorithm, which is loaded to the A-NPU before invocation. The multi-layer perceptron consists of layers of neurons, where the inputs of each layer are the outputs of the previous layer. The ANU starts from the input layer and performs the computations of the neurons layer by layer. The Input Buffer always contains the inputs to the neurons, either coming from the processor or from the previous layer computation. The Output Buffer, which is a single entry buffer, collects the outputs of the ANUs. When all of its columns are computed, the results are pushed back to the Input Buffer to enable calculation of the next layer. The Row Selector determines which entry of the input buffer will be fed to the ANUs. The output of the ANUs will be written to a single-entry output buffer. The Column Selector determines which column of the output buffer will be written by the ANUs. These selectors are FIFO buffers whose values are part of the preloaded A-NPU configuration. All the buffers are digital SRAM structures.

Each ANU has eight inputs. As depicted in Figure 4, each A-NPU is augmented with a dedicated weight buffer that stores the 8-bit weights. The weight buffers simultaneously feed the weights to the ANUs. The weights and the order in which they are fed to the ANUs are part of the A-NPU configuration. The Input Buffer and Weight Buffers synchronously provide the inputs and weights for the ANUs based on a pre-loaded order.

A-NPU configuration. During code generation, the compiler produces an A-NPU configuration that constitutes the weights and the schedule. The static A-NPU scheduling algorithm first assigns an order to the neurons. This determines the order in which the neurons will be computed on the ANUs. Then, the scheduler takes the following steps for each layer of the neural network: (1) Assign each neuron to one of the ANUs. (2) Assign an order to neurons. (3) Assign an order to the weights. (4) Generate the order for inputs to be fed to the ANUs. (5) Generate the order in which the outputs will be written to the Output Buffer. The scheduler also assigns a unique order for the inputs and outputs of the neural network in which the processor will communicate data with the A-NPU

4. Compilation for Analog Acceleration

As Figure 1 illustrates, the compilation for A-NPU execution consists of three stages: (1) profile-driven data collection, (2) training for a limited-precision A-NPU, and (3) code generation for hybrid analog-digital execution. In the profile-driven data collection stage, the compiler instruments the application to collect the inputs and outputs of approximable functions. The compiler then runs the application with representative inputs and collects the inputs and their corresponding outputs. These input-output pairs constitute the training data. Section 3 briefly discussed ISA extensions and code generation. While compilation stages (1) and (3) are similar to the techniques presented for a digital implementation [12], the training phase is unique to an analog approach, accounting for analog-imposed, topology restrictions and adjusting weight selection to account for limited-precision computation.

Hardware/software interface for exposing analog circuits to the compiler. As we discussed in Section 2, we expose the following analog circuit restrictions to the compiler through a hardware/software interface that captures the following circuit characteristics: (1) input bit-width limitations, (2) weight bit-width limitations, (3) limited number of inputs to each analog neuron (topology restriction), and (4) the non-ideal shape of the analog sigmoid. The compiler internally constructs a high-level model of the circuit based on these limitations and uses this model during the neural topology search and training with the goal of limiting the impact of inaccuracies due to an analog implementation.

Training for limited bit widths and analog computation. Traditional training algorithms for multi-layered perceptron neural networks use a gradient descent approach to minimize the average network error, over a set of training input-output pairs, by backpropagating the output error through the network and iteratively adjusting the weight values to minimize that error. Traditional training techniques, however, that do not consider limited-precision inputs, weights, and outputs perform poorly when these values are saturated to adhere to the bit-width requirements that are feasible for an implementation in the analog domain. Simply limiting weight values during training is also detrimental to achieving quality outputs because the algorithm does not have sufficient precision to converge to a quality solution.

Table 1: The evaluated benchmarks, characterization of each offloaded function, training data, and the trained neural network.

Benchmark Name	Description	Type	# of Function Calls	# of Loops	# of Ifs/ elses	# of x86-64 Instructions	Evaluation Input Set	Training Input Set	Neural Network Topology	Fully Digital NN MSE	Analog NN MSE (8-bit)	Application Error Metric	Fully Digital Error	Analog Error
blackscholes	Mathematical model of a financial market	Financial Analysis	5	0	5	309	4096 Data Point from PARSEC	16384 Data Point from PARSEC	6 -> 8 -> 8-> 1	0.000011	0.00228	Avg. Relative Error	6.02%	10.2%
fft	Radix-2 Cooley-Tukey fast fourier	Signal Processing	2	0	0	34	2048 Random Floating Point Numbers	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	0.00194	Avg. Relative Error	2.75%	4.1%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	4	0	0	100	10000 (x, y) Random Coordinates	10000 (x, y) Random Coordinates	2 -> 8 -> 2	0.000341	0.00467	Avg. Relative Error	6.2%	9.4%
jmeint	Triangle intersection detection	3D Gaming	32	0	23	1,079	10000 Random Pairs of 3D Triangle Coordinates	10000 Random Pairs of 3D Triangle Coordinate	18 -> 32 -> 8 -> 2	0.05235	0.06729	Miss Rate	17.68%	19.7%
jpeg	JPEG encoding	Compression	3	4	0	1,257	220x200-Pixel Color Image	Three 512x512-Pixel Color Images	64 -> 16 -> 8 -> 64	0.0000156	0.0000325	Image Diff	5.48%	8.4%
kmeans	K-means clustering	Machine Learning	1	0	0	26	220x200-Pixel Color Image	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00752	0.009589	Image Diff	3.21%	7.3%
sobel	Sobel edge detector	Image Processing	3	2	1	88	220x200-Pixel Color Image	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.000782	0.00405	Image Diff	3.89%	5.2%

To incorporate bit-width limitations into the training algorithm, we use a customized continuous-discrete learning method (CDLM) [8]. This approach takes advantage of the availability of full-precision computation at training time and then adjusts slightly to optimize the network for errors due to limited-precision values. In an initial phase, CDLM first trains a fully-precise network according to a standard training algorithm, such as backpropagation [24]. In a second phase, it discretizes the input, weight, and output values according the the exposed analog specification. The algorithm calculates the new error and backpropagates that error through the fully-precise network using full-precision computation and updates the weight values according to the algorithm also used in stage 1. This process repeats, backpropagating the 'discrete' errors through a precise network. The original CDLM training algorithm was developed to mitigate the impact of limited-precision weights. We customize this algorithm by incorporating the input bit-width limitation and the output bit-width limitation in addition to limited weight values. Additionally, this training scheme is advantageous for an analog implementation because it is general enough to also make up for errors that arise due to an analog implementation, such as a non-ideal sigmoid function and any other analog non-ideality that behaves consistently.

Training with topology restrictions. Conventional multi-layered perceptron networks are fully connected, i.e. the output of each neuron in one layer is routed to the input of each neuron in the following layer. However, analog range limitations restrict the number of inputs that can be computed in a neuron (eight in our design). Consequently, network connections must be limited, and in many cases, the network can not be fully connected.

We impose the circuit restriction on the connectivity between the neurons during the topology search and we use a simple algorithm guided by the mean-squared error of the network to determine the best topology given the exposed restriction. The error evaluation uses a typical cross-validation approach: the compiler partitions the data collected during profiling into a *training set*, 70% of the data, and a *test set*, the remaining 30%. The topology search algorithm trains many different neural-network topologies using the training set and chooses the one with the highest accuracy on the test set and the lowest latency on the A-NPU hardware (prioritizing accuracy). The space of possible topologies is large, so we restrict the search to neural networks with at most two hidden layers. We also limit the number of neurons per hidden layer to powers of two up to 32.

To further improve accuracy, and compensate for topology-restricted networks, we utilize a Resilient Back Propagation (RPROP) [16] training algorithm as the base training algorithm in our CDLM framework. During training, instead of updating the weight values based on the backpropagated error (as in conventional backpropagation [24]), the RPROP algorithm increases or decreases the weight values by a predefined value based on the sign of the error. Our investigation showed that RPROP significantly outperforms conventional backpropagation for the selected network topologies, requiring only half of the number of training epochs as backpropagation to converge on a quality solution. The main advantage of the application of RPROP training to an analog approach to neural computing is its robustness to the sigmoid function and topology restrictions imposed by the analog design. Our RPROP-based, customized CDLM training phase requires 5000 training epochs, with the analog-based CDLM phase adding roughly 10% to the training time of the baseline training algorithm.

5. Evaluations

Cycle-accurate simulation and energy modeling. We use the MARSSx86 x86-64 cycle-accurate simulator [23] to model the performance of the processor. The processor is modeled after a single-core Intel Nehalem to evaluate the performance benefits of A-NPU acceleration over an aggressive out-of-order architecture. We extended the simulator to include ISA-level support for A-NPU queue and dequeue instructions. We

Table 2: Error with a floating point D-NPU, A-NPU with ideal sigmoid, and A-NPU with non-ideal sigmoid.

	blackscholes	fft	inversek2j	jmeint	jpeg	kmeans	sobel
Floating Point D-NPU	6.0%	2.7%	6.2%	17.6%	5.4%	3.2%	3.8%
A-NPU + Ideal Sigmoid	8.4%	3.0%	8.1%	18.4%	6.6%	6.1%	4.3%
A-NPU	10.2%	4.1%	9.4%	19.7%	8.4%	7.3%	5.2%

also augmented MARSSx86 with a cycle-accurate simulator for our A-NPU design and an 8-bit, fixed-point D-NPU with eight processing engines (PEs) as described in [12]. We use GCC v4.7.3 with -o3 to enable compiler optimization. The baseline in our experiments is the benchmark run solely on the processor without neural transformation. We use McPAT [19] for processor energy estimations. We model the energy of an 8-bit, fixed-point D-NPU using results from McPAT, CACTI 6.5 [21], and [13] to estimate its energy. Both the D-NPU and the processor operate at 3.4GHz, while the A-NPU is clocked at one third of the digital clock frequency, 1.1GHz at 1.2 V, to achieve acceptable accuracy[1].

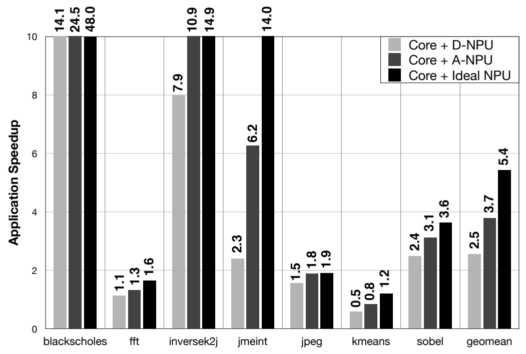
Circuit design for ANU. We implemented the 8-bit, 8-input ANU in the Cadence Analog Design Environment using predictive technology models at 45 nm [5]. We ran detailed Spectre spice simulations to understand circuit behavior and measure ANU energy consumption. We used CACTI to estimate energy of the A-NPU buffers.

Benchmarks. We use the benchmarks in [12] and add one more, blackscholes.

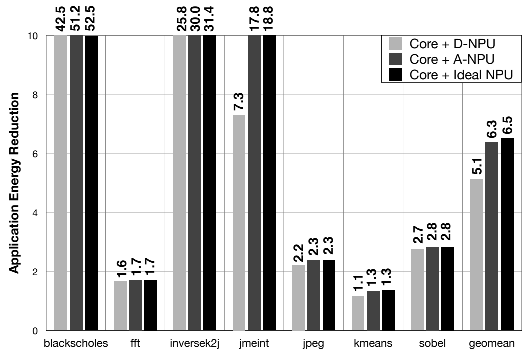
Whole application speedup and energy savings. Figure 5 shows the whole application speedup and energy savings when the processor is augmented with an 8-bit, 8-PE D-NPU, our 8-ANU A-NPU, and an ideal NPU, which takes zero cycles and consumes zero energy. Figure 5c shows the percentage of dynamic instructions subsumed by the neural transformation of the candidate code. The results show, following the Amdahl's Law, that the larger the number of dynamic instructions subsumed, the larger the benefits from neural acceleration. Geometric mean speedup and energy savings with an A-NPU is 3.7× and 6.3× respectively, which is 48% and 24% better than an 8-bit, 8-PE NPU. Among the benchmarks, kmeans sees slow down with D-NPU and A-NPU-based acceleration. All benchmarks benefit in terms of energy. The speedup with A-NPU acceleration ranges from 0.8× to 24.5×. The energy savings range from 1.1× to 51.2×.

Application error. Table 2 shows the application-level errors with a floating point D-NPU, A-NPU with ideal sigmoid

[1]**Processor:** Fetch/Issue Width: 4/5, INT ALUs/FPUs: 6/6, Load/Store FUs: 1/1, ROB Entries: 128, Issue Queue Entries: 36, INT/FP Physical Registers: 256/256, Branch Predictor: Tournament 48 KB, BTB Sets/Ways: 1024/4, RAS Entries: 64, Load/Store Queue Entries: 48/48, Dependence Predictor: 4096-entry Bloom Filter, ITLB/DTLB Entries: 128/256 **L1:** 32 KB Instruction, 32 KB Data, Line Width: 64 bytes, 8-Way, Latency: 3 cycles **L2:** 256 KB, Line Width: 64 bytes, 8-Way, Latency: 6 cycles **L3:** 2 MB, Line Width 64 bytes, 16-Way, Latency: 27 cycles **Memory Latency:** 50 ns



(a) Whole application speedup.



(b) Whole application energy saving.

	blackscholes	fft	inversek2j	jmeint	jpeg	kmeans	sobel
Percentage Instructions Subsumed	97.2%	67.4%	95.9%	95.1%	56.3%	29.7%	57.1%

(c) % dynamic instructions subsumed.

Figure 5: Whole application speedup and energy saving with D-NPU, A-NPU, and an Ideal NPU that consumes zero energy and takes zero cycles for neural computation.

and our A-NPU which incorporates non-idealities of the analog sigmoid. Except for jmeint, which shows error above 10%, all of the applications show error less than or around 10%. Application average error rates with the A-NPU range from 4.1% to 10.2%. This quality-of-result loss is commensurate with other work on quality trade-offs use digital techniques. Truffle [11] and EnerJ [26] shows similar error (3–10%) for some applications and much greater error (above 80%) for others in a moderate configuration. Green [2] has error rates below 1% for some applications but greater than 20% for others. A case study [20] explores manual optimizations of the x264 video encoder that trade off 0.5–10% quality loss.

6. Related Work

General-purpose approximate computing. A growing body of work has explored relaxing the abstraction of full accuracy at the circuit and architecture level for gains in performance, energy, and resource utilization [9, 11, 25, 6, 22, 12]. These circuit and architecture studies, although proven successful, are limited to purely digital techniques. We explore how a mixed-signal, analog-digital approach can go beyond what digital approximate techniques offer.

Analog and digital neural hardware. There is an extensive body of work on hardware implementations of neural networks both in the digital [10, 30] and the analog [4, 18] domain. Recent work has proposed higher-level abstractions for implementation of neural networks [15]. Other work has examined fault-tolerant hardware neural networks [14, 29]. In particular, Temam [29] uses datasets from the UCI machine learning repository to explore fault tolerance of a hardware neural network design. In contrast, our compilation, neural-network selection/training framework, and architecture design aim at applying neural networks to general-purpose code written in familiar programming models and languages, not explicitly written to utilize neural networks directly.

Neural-based code acceleration. A recent study [7] shows that a number of applications can be manually reimplemented with explicit use of various kinds of neural networks. That study did not prescribe a programming workflow, nor a preferred hardware architecture. More recent work exposes analog spiking neurons as primitive operators [3]. This work devises a new programming model that allows programmers to express digital signal-processing applications as a graph of analog neurons and automatically maps the expressed graph to a tiled analog, spiking-neural hardware. The work in [3] is restricted to the domain of applications whose input are real-world signals that should be encoded as pulses. Our approach intends to address the long-standing challenges of utilizing analog computation (programmability and generality) by not imposing domain-specific limitations, and by providing analog circuitry that is integrated with a conventional, digital processor without require a new programming paradigm.

7. Conclusions

Analog circuits inherently trade accuracy for very attractive energy-efficiency gains. However, it is challenging to utilize them in a way that is both programmable and generally useful. The transformation of general-purpose, approximable code to a neural model, as well as the use of neural accelerators, provide an avenue for realizing the benefits of analog computation by taking advantage of the fixed-function qualities of a neural network while targeting traditionally-written, generally-approximable code. We presented a complete solution on using analog neural networks for accelerating approximate applications, from circuits to compilers design. A very important insight from this work is that it is crucial to expose analog circuit characteristics to the compilation and neural network training phase. Our analog neural acceleration provides whole application speedup of 3.3× and and energy savings of 12.1× with quality loss less than 10% for all except one benchmark.

References

[1] P. E. Allen and D. R. Holberg, *CMOS Analog Circuit Design*, 2nd ed. Oxford University Press, 2002.

[2] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[3] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, 2013.

[4] B. E. Boser, E. Säckinger, J. Bromley, Y. L. Cun, L. D. Jackel, and S. Member, "An analog neural network processor with programmable topology," *J. Solid-State Circuits*, vol. 26, no. 12, December 1991.

[5] Y. Cao, "Predictive technology models," 2013. Available: http://ptm.asu.edu

[6] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *DATE*, 2006.

[7] 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 *IISWC*, 2012.

[8] F. Choudry, E. Fiesler, A. Choudry, and H. J. Caulfield, "A weight discretization paradigm for optical neural networks," in *ICOE*, 1990.

[9] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[10] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, "Neural network stream processing core (NnSP) for embedded systems," in *ISCAS*, 2006.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[13] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Trans. Comput.*, vol. 60, no. 7, 2011.

[14] A. Hashmi, H. Berry, O. Temam, and M. Lipasti, "Automatic abstraction and fault tolerance in cortical microarchitectures," in *ISCA*, 2011.

[15] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, "A case for neuromorphic ISAs," in *ASPLOS*, 2011.

[16] C. Igel and M. Hüsken, "Improving the RPROP learning algorithm," in *NC*, 2000.

[17] D. A. Johns and K. Martin, *Analog Integrated Circuit Design*. John Wiley and Sons, Inc., 1997.

[18] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware spiking neurons design: Analog or digital?" in *IJCNN*, 2012.

[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[20] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[21] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[22] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010.

[23] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *DAC*, 2011.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986, vol. 1.

[25] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *MICRO*, 2013.

[26] 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 *PLDI*, 2011.

[27] J. Schemmel, J. Fieres, and K. Meier, "Wafer-scale integration of analog neural networks," in *IJCNN*, 2008.

[28] S. M. Tam, B. Gupta, H. A. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in *Systems, Man, and Cybernetics (SMC)*, 1990.

[29] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ISCA*, 2012.

[30] J. Zhu and P. Sutton, "FPGA implementations of neural networks: A survey of a decade of progress," in *FPL*, 2003.