DOI:10.1145/2589750

Neural Acceleration for General-Purpose Approximate Programs

By Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger

Abstract

As improvements in per-transistor speed and energy efficiency diminish, radical departures from conventional approaches are needed to continue improvements in the performance and energy efficiency of general-purpose processors. One such departure is approximate computing, where error in computation is acceptable and the traditional robust digital abstraction of near-perfect accuracy is relaxed. 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—error—and trades the accuracy of computation for gains in both energy and performance. Techniques to harvest large savings from small errors have proven elusive. This paper describes a new approach that uses machine learning-based transformations to accelerate approximation-tolerant programs. The core idea is to train a learning model how an approximable *region* of code—code that can produce imprecise but acceptable results—behaves and replace the original code region with an efficient computation of the learned model. We use neural networks to learn *code* behavior and approximate it. We describe the *Parrot algorithmic transformation*, which leverages a simple programmer annotation ("approximable") to transform a code region from a von Neumann model to a neural model. After the learning phase, the compiler replaces the original code with an invocation of a low-power accelerator called a *neural processing unit* (NPU). The NPU is tightly coupled to the processor pipeline to permit profitable acceleration even when small regions of code are transformed. Offloading approximable code regions to NPUs is faster and more energy efficient than executing the original code. For a set of diverse applications, NPU acceleration provides whole-application speedup of 2.3× and energy savings of 3.0× on average with average quality loss of at most 9.6%. NPUs form a new class of accelerators and show that significant gains in both performance and efficiency are achievable when the traditional abstraction of near-perfect accuracy is relaxed in general-purpose computing.

1. INTRODUCTION

It is widely understood that energy efficiency now fundamentally limits microprocessor performance gains. CMOS scaling is no longer providing gains in efficiency commensurate with transistor density increases.[7, 15] As a result, both the semiconductor industry and the research community are increasingly focusing on specialized accelerators, which can provide large gains in efficiency and performance by restricting the workloads that benefit. Recent work has quantified three orders of magnitude of difference in efficiency between general-purpose processors and ASICs.[14] The community is facing an "iron triangle" in this respect; we can choose any two of performance, energy efficiency, and generality at the expense of the third. Before the traditional trend of transistor scaling—Dennard scaling[5]—broke down, we were able to improve all three on a consistent basis for decades. In this post Dennard scaling era, solutions that improve performance and efficiency while retaining as much generality as possible are highly desirable; hence the exploding interest in GPGPUs and FPGAs. Such programmable accelerators exploit some characteristic of an application domain to achieve efficiency gains at the cost of generality. FPGAs, for example, exploit copious, fine-grained, and irregular parallelism while GPUs exploit many threads and data-level SIMD-style parallelism. Whether an application can use an accelerator effectively depends on the degree to which it exhibits the accelerator's required characteristics.

Tolerance to approximation is one such program characteristic. A growing body of recent work[2, 4, 8, 19, 26, 27] has focused on *approximation* as a strategy for improving efficiency. Large classes of applications can tolerate small errors in their outputs with no discernible loss in their *quality of result* (QoR). These applications are common in mobile, embedded, and server systems and fall into four broad categories:

1. *Applications with analog inputs*. This category includes image processing, sensor data processing, voice recognition, etc., that operate on noisy real-world data. They are inherently resilient to some noise and can handle an "extra noise" resulting from approximation.
2. *Applications with analog output*. These applications comprise multimedia, image rendering, sound synthesis, etc. Their output is intended for human perception and can inherently tolerate errors imperceptible to users.
3. *Applications with no unique answer*. This class of applications includes web search, machine learning, autonomous agents, etc., which do not offer a unique answer and multiple possible answers are acceptable.

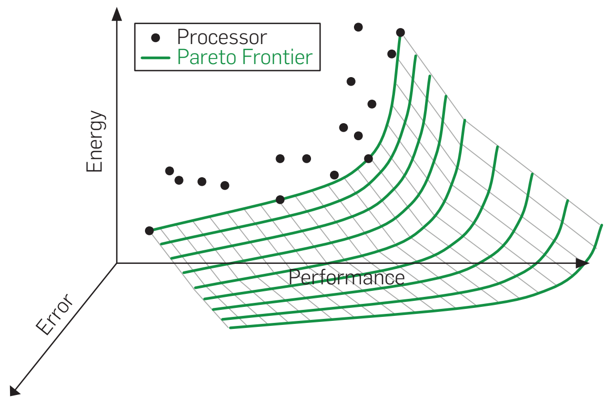
A previous version of this article appeared in *Proceedings of the 45th Symposium on Microarchitecture* (December 2012). Parts of this article appeared in *IEEE Micro Top Picks from the Computer Architecture Conferences of 2012* (May/June 2013).

4. *Iterative and convergent applications*. This category includes applications such as data analytics and numerical computations that iteratively process large amounts of data. They often sample data, stop the convergence procedure early, or apply approximate heuristics. Thus, these applications can naturally benefit from approximation techniques.

For these classes of applications, trading off computation accuracy can potentially lead to larger gains in performance and efficiency. One may visualize these trade-offs as finding the Pareto-optimal points in the processor design space, as shown in Figure 1. Traditionally, for any set of workloads, the set of possible processor implementations may be plotted, with energy efficiency on one axis and performance on the other, and the best implementations residing on the two-dimensional frontier. When approximation is supported, the degree of permissible error represents a third axis. The Pareto surface in this three-dimensional space represents the best points of performance, efficiency, and error. However, this surface is not yet well understood.

This paper defines some new points on this Pareto surface by developing a new class of programmable accelerators that exploit approximation for better performance and energy efficiency. The core idea is to *learn* how a region of approximable code behaves and *automatically* replace the original code with an efficient computation of the learned model. This approach contrasts with previous work on approximate computation that extends conventional microarchitectures to support selective approximate execution, incurring instruction bookkeeping overheads[4, 8, 19] or requires vastly different programming paradigms.[2, 21] Like emerging flexible accelerators,[11, 12, 29] our technique automatically offloads code segments from programs written in mainstream languages. However, unlike prior work, it leverages changes in the semantics of the offloaded code and the nature of computation. Such changes are possible because the transformed code region is approximable and can tolerate small errors.

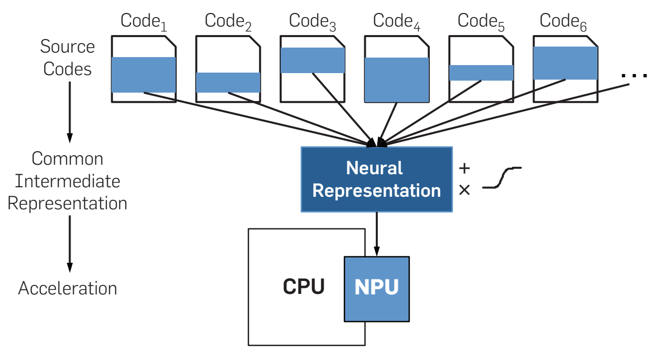
Figure 1. Adding the dimension of error to the design space of general-purpose processors changes the problem of finding the Pareto frontier to finding the Pareto surface. Navigating this three-dimensional space, finding, and understanding this Pareto surface is a fascinating research direction.



NPU-enabled systems rely on our learning-based algorithmic transformation that we refer to as the Parrot transformation[a]. This transformation converts regions of approximable general-purpose code into a neural representation—specifically, multilayer perceptrons—at compile time. At run time, while the processor executes the program, it invokes the NPU instead of running the original region of code. NPUs result in large performance and efficiency gains, since they subsume a region of code, eliminating nearly all of the instruction fetch, decode, etc., overheads that would have been incurred if that region was executed on the processor. These overheads are replaced by a single, efficient invocation of neural hardware that exploits hard-wired control for additional efficiency. As illustrated in Figure 2, NPU acceleration provides improved generality over task-specific accelerators, as the Parrot transformation converts many distinct code patterns into a common representation that can be run on a single physical accelerator. In fact, our Parrot algorithmic transformation replaces unstructured serial code with a neural network that has structured fine grain parallelism that is executed on an efficient statically scheduled hardware structure, the NPU. Therefore, converting diverse regions of code to the common neural representation can lead to significant performance and efficiency gains because neural networks consist of simple, regular, and parallel operations.

We show that using neural networks to replace regions of imperative code is both feasible and profitable by accelerating a diverse range of applications, including FFT, gaming, clustering, and vision algorithms (Section 6). These applications do not belong to the class of modeling and prediction tasks that typically use neural networks. For each application, we apply the transformation on a single approximable function that dominates the program's execution time. NPU acceleration provides 2.3× average whole-application speedup and 3.0× average energy savings for these benchmarks with average accuracy greater than 90% in all cases.

Figure 2. The Parrot algorithmic transformation converts different regions of code to a common neural intermediate representation. Neural networks as a common representation enable acceleration of diverse applications using a single physical NPU.



[a] We named our algorithmic transformation, the Parrot transformation because its output is a learning model that mimics the original region of code.

The Parrot algorithmic transformation and the NPU acceleration bridge von Neumann and neural models of computing. These techniques make neural hardware programmable via conventional programming languages and extend their use beyond prediction and modeling to accelerating general-purpose code. The results from this paper show that when the traditional abstraction of near-perfect accuracy is relaxed, different models of computing can be merged to obtain large gains in performance and efficiency.

2. OVERVIEW

As depicted in Figure 3, the *Parrot transformation* is an algorithmic transformation that converts regions of imperative code to neural networks. Because neural networks expose considerable parallelism and consist of simple operations, they can be efficiently accelerated using dedicated hardware. Therefore, the Parrot transformation can yield significant performance and energy improvements. The transformation uses a training-based approach to produce a neural network that approximates the behavior of candidate code. A transformed program runs primarily on the main core and invokes an auxiliary hardware structure, the NPU, to perform neural evaluation instead of executing the replaced code. Figure 3 shows an overview of the Parrot algorithmic transformation, which has three key phases: programming, in which the programmer marks code regions to be transformed; compilation, in which the compiler selects and trains a suitable neural network and replaces the original code with a neural network invocation; and execution.

Programming. The Parrot transformation starts with the programmer identifying candidate code regions as approximable. Because tolerance of approximation is a semantic property, it is the programmer's responsibility to select code whose approximate execution would not compromise the overall reliability of the application. This requirement is a common practice in the approximate computing literature.[4, 8, 26]

Compilation. Once the source code is annotated, as shown in Figure 3, the compiler applies the Parrot transformation in three steps: (1) code observation, (2) neural network selection and training, and (3) binary generation.

Training neural networks for any task requires a collection of input–output pairs that capture the task's function. Therefore, in the code observation step, the compiler observes the behavior of the candidate code region by logging its inputs and outputs. This step is similar to profiling. The compiler instruments the program with probes on the inputs and outputs of the candidate functions. Then, the instrumented program is run using representative input sets such as those from a test suite. The probes log the inputs and outputs of the candidate functions. The logged input–output pairs constitute the training and validation data for the next step.

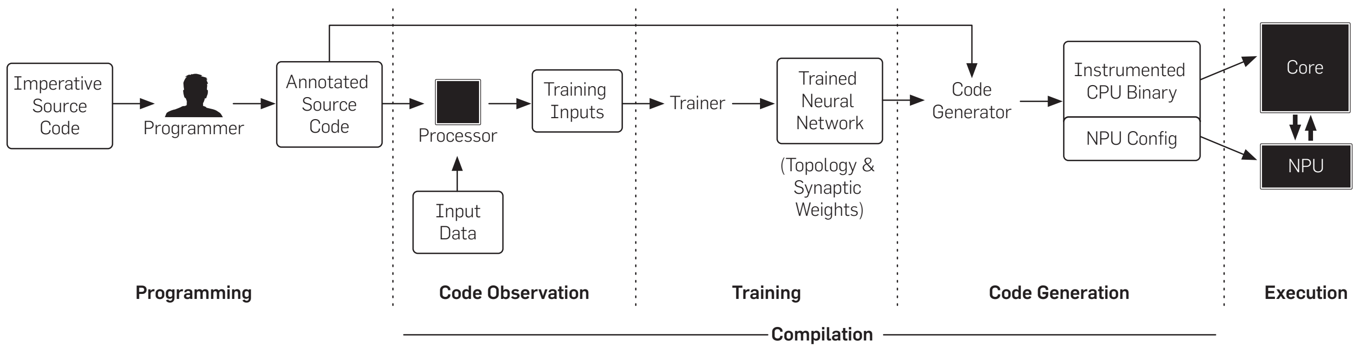
The compiler uses the collected input–output data to configure and train a neural network that mimics the candidate region. The compiler must find the simplest topology of the neural network that provides acceptable error, for which more complex networks would provide diminishing returns in QoR. The compiler also needs to find the synaptic weights of the network. It uses the backpropagation algorithm[25] coupled with a topology search to configure and train the neural network.

The final step of the Parrot transformation is code generation. The compiler first generates a configuration for the NPU that implements the trained neural network. Then, the compiler replaces each call to the original function with a series of special instructions that invoke the NPU, sending the inputs and receiving the computed outputs. The NPU configuration and invocation is performed through ISA extensions that are added to the core.

Execution. During deployment, the transformed program begins execution on the main core and configures the NPU. Throughout execution, the NPU is invoked to perform a neural network evaluation in lieu of executing the original code region. The NPU is integrated as a tightly coupled accelerator in the processor pipeline. Invoking the NPU is faster and more energy efficient than executing the original code region, so the program as a whole is accelerated.

As Figure 4 shows, many NPU implementations are feasible, from all-software execution to specialized analog circuits. Because the Parrot transformation's effectiveness rests on the efficiency of neural network evaluation, it is essential that invoking the NPU be fast and low power. Therefore, we describe a high-performance hardware NPU design based on a digital neural network ASIC and architecture support to facilitate low-latency NPU invocations.

Figure 3. The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.



3. PROGRAMMING MODEL

The Parrot transformation starts with the programmer identifying candidate code regions. This section discusses these criteria as well as the concrete language interface exposed to the programmer. After the candidate regions are identified, the Parrot transformation is fully automated.

3.1. Code region criteria

Candidate code for the Parrot transformation must satisfy three criteria: it must be frequently executed (i.e., a "hot" function); it must tolerate imprecision in its computation; and it must have well-defined inputs and outputs.

Hot code. Like any acceleration technique, the Parrot transformation should replace hot code. The Parrot transformation can be applied to a wide range of code from small functions to entire algorithms. The code region can contain function calls, loops, and complex control flow whose cost can be elided by the Parrot transformation. When applied to smaller regions of code, the overhead of NPU invocation needs to be low to make the transformation profitable. A traditional performance profiler can reveal hot code.

For example, edge detection is a widely applicable image processing computation. Many implementations of edge detection use the Sobel filter, a 3 × 3 matrix convolution that approximates the image's intensity gradient. As the bottom box in Figure 5a shows, the local Sobel filter computation (the `sobel` function) is executed many times during edge detection, so the convolution is a hot function in the overall algorithm and a good candidate for the Parrot transformation.

Approximability. Code regions identified for the Parrot transformation will behave approximately during execution. Therefore, programs must incorporate application-level tolerance of imprecision. This requires the programmer to ensure that imprecise results from candidate regions will not cause catastrophic failures. As prior work on approximate programming[1, 4, 19, 26, 27] has shown, it is not difficult to deem regions approximable.

Beyond determining that a code region may safely produce imprecise results, the programmer need not reason about the mapping between the code and a neural network. While neural networks are more precise for some functions than they are for others, we find that they can accurately mimic many functions from real programs (see Section 6). Intuitively, however, they are less likely to effectively approximate chaotic functions, in which even large training sets can fail to capture enough of the function's behavior to generalize to new inputs. However, the efficacy of neural network approximation can be assessed empirically. The programmer should annotate all approximate code; the compiler can then assess the accuracy of a trained neural network in

Figure 4. Design space of NPU implementations. This work focuses on a precise digital ASIC design.

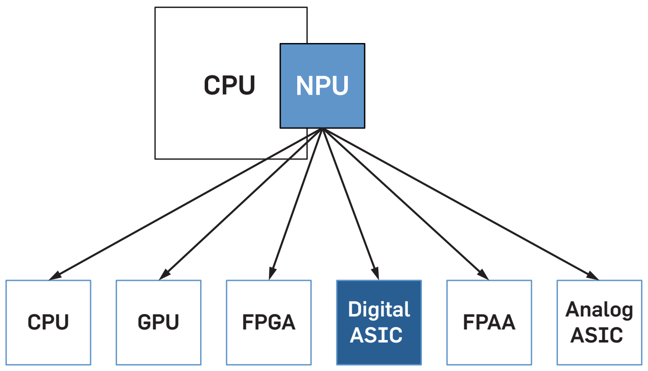
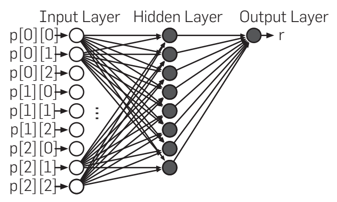


Figure 5. Three stages in the transformation of an edge detection algorithm using the Sobel filter: (a) original implementation of the Sobel filter, (b) sobel function transformed to a 9 → 8 → 1 NN, and (c) Parrot transformed code; NPU invocation replaces the function call.

```
float sobel [[approximable]] (float[3][3] p){
  float x, y, r;
  x = (p[0][0] + 2 * p[0][1] + p[0][2]);
  x += (p[2][0] + 2 * p[2][1] + p[2][2]);
  y = (p[0][2] + 2  * p[1][2] + p[2][2]);
  y += (p[0][0] + 2 * p[1][1] + p[2][0]);
  r = sqrt (x * x + y * y);
  if (r >= 0.7071) r = 0.7070;
  return r;
}
```

```
void edgeDetection (Image& srcImg, Image& dstImg){
  float [3][3] p; float pixel;
  for (int y = 0; y < srcImg.height; ++y)
    for (int x = 0; x < srcImg.width; ++x)
      srcImg.toGrayeScale(x,y);
  for (int y = 0; y < srcImg.height; ++y)
    for (int x = 0; x < scrImg.width; ++x){
      p = srcImg.build3x3Window(x, y);
      pixel = sobel(p);
      dstImg.setPixel(x, y, pixel);
    }
}
```

(a)



(b)

```
void edgeDetection (Image& srcImg, Image& dstImg){
  float [3][3] p; float pixel;
  for (int y = 0; y < srcImg.height; ++y)
    for (int x = 0; x < srcImg.width; ++x)
      srcImg.toGrayeScale(x, y);
  for (int y = 0; y < srcImg.height; ++y)
    for (int x = 0; x < scrImg.width; ++x){
      p = srcImg.build3x3Window(x, y);
      NPU_SEND(p[0][0]); NPU_SEND(p[0][1]); NPU_SEND(p[0][2]);
      NPU_SEND(p[1][0]); NPU_SEND(p[1][1]); NPU_SEND(p[1][2]);
      NPU_SEND(p[2][0]); NPU_SEND(p[2][1]); NPU_SEND(p[2][2]);
      NPU_RECEIVE(pixel);
      dstImg.setPixel(x, y, pixel);
    }
}
```

(c)

replacing each function and select only those functions for which neural networks are a good match.

In the Sobel filter example, parts of the code that process the pixels can be approximated. The code region that computes pixel addresses and builds the window for the sobel function (line 8 in the bottom box of Figure 5a) needs to be precise to avoid memory access violations. However, the sobel function, which estimates the intensity gradient of a pixel, is fundamentally approximate. Thus, approximate execution of this function will not result in catastrophic failure and, moreover, is unlikely to cause major degradation of the overall edge detection quality. These properties make the sobel function a suitable candidate region for approximate execution.

Well-defined inputs and outputs. The Parrot transformation replaces a region of code with a neural network that has a fixed number of inputs and outputs. Therefore, it imposes two restrictions on the code regions that can feasibly be replaced. First, the inputs to and outputs from the candidate region must be of a fixed size known at compile time. For example, the code may not dynamically write an unbounded amount of data to a variable-length array. Second, the code must not cause side effects via system calls. These two criteria can be checked statically.

The sobel function in Figure 5a complies with these requirements. It takes nine statically identifiable floating-point numbers as input, produces a single output, and has no side effects.

3.2. Annotation

In this work, we apply the Parrot transformation to entire functions. To identify candidate functions, the programmer marks them with an annotation (e.g., using C++11 [[approximble]] syntax) as shown in Figure 5a. The programmer is responsible for ensuring that the function has well-defined inputs and outputs. All the inputs are in the argument list and all the outputs are part of the return value. Each argument type and the return type must have a fixed size; however, they may have multiple elements, for example, fixed-size array or a record. If any of these types is a pointer type, it must point to a fixed-size value; this referenced value is then considered the neural network input or output rather than the pointer itself. If the function needs to return multiple values, it can return a fixed-size array or a C struct. After the programmer annotates the candidate functions, the Parrot transformation is completely automatic and transparent: no further programmer intervention is necessary.

Like prior work on approximate computing, we acknowledge that some programmer guidance is essential when identifying error-tolerant code.[1, 4, 8, 19, 26] Tolerance to approximation is an inherently application-specific property. While we find that the simple explicit function annotations are straightforward to apply (see Section 6), static analysis techniques could be used to further simplify the annotation process and significantly automate it.

4. COMPILATION WORKFLOW

Once the program has been annotated, the compilation workflow implements the Parrot transformation in three steps: observation, training, and instrumented binary generation.

4.1. Code observation

Training neural networks for any task requires a collection of input–output pairs that capture the task's function. Therefore, in the first phase, the compiler collects input–output pairs for the target code that reflect real program executions. This in-context observation allows the compiler to train the neural network on a realistic data set. The compiler produces an instrumented binary for the source program that includes probes on the input and output of the annotated function. Each time the candidate function executes, the probes record its inputs and outputs. The program is run repeatedly using test inputs. The output of this phase is a training data set: each input–output pair represents a sample for the training algorithm.

The observation phase resembles the profiling runs used in profile-guided compilation. Specifically, it requires representative test inputs for the application. The inputs may be part of an existing test suite or randomly generated. In many cases, a small number of application test inputs are sufficient to train a neural network because the candidate function is executed many times in a single application run.

4.2. Training

The compiler uses the training data to produce a neural network that replaces the original function. There are a variety of types of artificial neural networks in the literature, but we narrow the search space to multilayer perceptrons (MLPs) due to their broad applicability.

The compiler uses the backpropagation algorithm[25] to train the neural network. Backpropagation is a gradient descent algorithm that iteratively adjusts the weights of the neural network according to each input–output pair.

Neural network topology selection. In addition to running backpropagation, this phase selects a network topology that balances accuracy and efficiency. An MLP consists of a fully connected set of neurons organized into layers: the input layer, any number of "hidden" layers, and the output layer. A larger, more complex network offers better accuracy potential but is likely to be slower and less power efficient than a small, simple neural network. The objective is to find the smallest neural network that achieves acceptable accuracy.

To choose the topology, we use a simple search algorithm guided by the mean squared error of the neural network when tested on an unseen subset of the observed data. The error evaluation uses a typical cross-validation approach: the compiler partitions the data collected during observation into a *training set*, 70% of the observed 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 NPU (prioritizing accuracy).

The output from this phase consists of a neural network topology—specifying the number of layers and the number of neurons in each layer—along with the weight for each neuron and the normalization range for each input and output.

Online training. The current system performs observation and training prior to deployment; an alternative design could train the neural network concurrently with in vivo operation. Online training could improve accuracy but would result in runtime overheads. To address these overheads, an online training system could offload neural network training and configuration to a remote server. With off-site training, multiple deployed application instances could centralize their training to increase input space coverage.

4.3. Code generation

After the training phase, the compiler generates an instrumented binary that runs on the core and invokes the NPU instead of calling the original function. The program configures the NPU when it is first loaded by sending the topology parameters and synaptic weights to the NPU. The compiler replaces the calls to the original function with special instructions that send the inputs to the NPU and collect the outputs from it.

5. ARCHITECTURE DESIGN FOR NPU ACCELERATION

Since candidate regions for the Parrot transformation can be fine grained, NPU invocation must be low-overhead to be beneficial. Ideally, the NPU should integrate tightly with the processor pipeline. The processor ISA also needs to be extended to allow programs to configure and invoke the NPU during execution.

5.1. ISA support for NPU acceleration

The NPU is a variable-delay, tightly coupled accelerator that communicates with the rest of the core via FIFO queues. The CPU–NPU interface consists of three queues: one for sending and retrieving the configuration, one for sending the inputs, and one for retrieving the neural network's outputs. The ISA is extended with four instructions to access the queues. These instructions assume that the processor is equipped with a single NPU; if the architecture supports multiple NPUs or multiple stored configurations per NPU, the instructions may be parameterized with an operand that identifies the target NPU.

- `enq.c %r`: enqueues the value of the register `r` into the config FIFO.
- `deq.c %r`: dequeues a configuration value from the config FIFO to the register `r`.
- `enq.d %r`: enqueues the value of the register `r` into the input FIFO.
- `deq.d %r`: dequeues the head of the output FIFO to the register `r`.

To set up the NPU, the program executes a series of `enq.c` instructions to send configuration parameters—number of inputs and outputs, network topology, and synaptic weights—to the NPU. The operating system uses `deq.c` instructions to save the NPU configuration during context switches. To invoke the NPU, the program executes `enq.d` repeatedly to send inputs to the configured neural network. As soon as all of the inputs of the neural network are enqueued, the NPU starts computation and puts the results in its output FIFO. The program executes `deq.d` repeatedly to retrieve the output values.

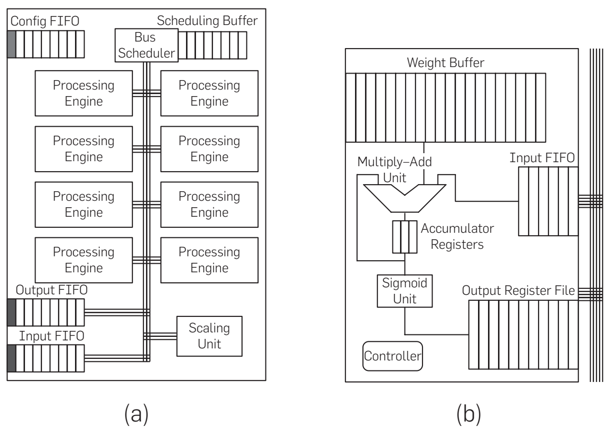
5.2. Neural processing unit

As Figure 4 illustrates, there are many possibilities for the implementation of the NPU itself. Neural networks have previously been implemented in software on the CPU or GPU,[13] on FPGAs,[30] in digital ASICs,[6] and even in analog circuitry or FPAAs.[17] As shown in Figure 6, we designed a reconfigurable digital NPU circuit that operates at the same voltage and frequency as the main core. More details on our NPU design can be found in Esmaeilzadeh et al.[9] This implementation represents a reasonable trade-off between efficiency and complexity. However, we believe that analog NPUs have significant potential and we plan to explore them in future work.

The Parrot transformation produces different neural network topologies for different code regions. Thus, our reconfigurable NPU accelerates the evaluation of a range of neural topologies. As shown in Figure 6, the NPU contains eight identical processing engines (PEs) and one scaling unit. The scaling unit scales the neural network's inputs and outputs if necessary using scaling factors defined in the NPU configuration process.

The PEs in the NPU are statically scheduled. The scheduling information is part of the configuration information for the NPU, which is based on the neural network topology derived during the training process. In the NPU's schedule, each neuron in the neural network is assigned to one of the eight PEs. The neural network's topology determines a static schedule for the timing of the PE computations, bus accesses, and queue accesses.

Figure 6. Reconfigurable 8-PE NPU: (a) 8-PE NPU, (b) single processing engine (PE), and (c) design parameters of the NPU.



Parameter	Configuration
Number of PEs	8
Bus Schedule FIFO	512 × 20-bit
Input FIFO	128 × 32-bit
Output FIFO	128 × 32-bit
Config FIFO	8 × 32-bit
PE Weight Cache	512 × 33-bit
PE Input FIFO	8 × 32-bit
PE Output Register File	8 × 32-bit
Sigmoid Unit LUT	128 × 32-bit
Multiply–Add Unit	32-bit Single Precision FP

(c)

The NPU stores the bus scheduling information in its circular scheduling buffer (shown in Figure 6). Figure 6b shows the internal structure of a single PE. Each PE performs the computation for all of its assigned neurons. Namely, because the NPU implements a sigmoid-activation MLP, each neuron computes its output as $y = \text{sigmoid}\left(\sum_i (x_i \times w_i)\right)$, where x_i is an input to the neuron and w_i is its corresponding weight. The weight buffer, a circular buffer, stores the weights. When a PE receives an input from the bus, it stores the value in its input FIFO. When the neuron weights for each PE are configured, they are placed into the weight buffer; the compiler-directed schedule ensures that the inputs arrive in the same order that their corresponding weights appear in the buffer. This way, the PE can perform multiply-and-add operations in the order the inputs enter the PE's input FIFO.

Since the computation of the neural network requires only simple operations that are limited to multiply–add and sigmoid lookup, the structure of the PEs is fairly simple. Furthermore, neural network computation exposes fine grain regular parallelism that we exploited in our NPU design by including multiple identical PEs. In fact, our Parrot algorithmic transformation replaces unstructured serial code with a neural network that has structured fine grain parallelism that is executed on an efficient statically scheduled hardware structure, NPU.

6. EVALUATION

To evaluate the effectiveness of the Parrot transformation, we apply it to several benchmarks from diverse application domains. For each benchmark, we identify a region of code that is amenable to the Parrot transformation. We evaluate whole-application speedup and energy savings using cycle-accurate simulation and a power model. We also examine the resulting trade-off in computation accuracy. We refer the reader to the original paper for more details on the evaluation.[9]

6.1. Benchmarks

Table 1 lists the approximation-tolerant applications from diverse domains that are used to evaluate the broad applicability of our technique. These benchmarks are all written in C. The application domains—signal processing, robotics, gaming, compression, machine learning, and image processing—are selected for their usefulness to general applications and tolerance to imprecision. The domains are commensurate with evaluations of previous work on approximate computing.[8, 19, 26, 27] To be able to assess the effect of the Parrot transformation perceptually, we selected a number of benchmarks that generate image outputs and compared the output image with and without the transformation. We did not reject any of the applications based on performance, energy, or accuracy shortfalls.

Table 1 also lists the input sets used for performance, energy, and accuracy assessment. These input sets are different from the ones used during the training phase of

Table 1. The benchmarks evaluated, details for each transformed function, input data, and the result of the Parrot transformation.

	Description	Type	Evaluation input set	No. of function calls	No. of loops	No. of ifs/ elses	No. of ×86-64 instructions	Training input set	Neural network topology	NN MSE	Error metric	Error
fft	Radix-2 Cooley-Tukey fast Fourier	Signal processing	2048 random floating-point numbers	2	0	0	34	32,768 random floating-point numbers	$1 \to 4 \to 4 \to 2$	0.00002	Average relative error	7.22%
inversek2j	Inverse kinematics for two-joint arm	Robotics	10,000 (*x*, *y*) random coordinates	4	0	0	100	10,000 (*x*, *y*) random coordinates	$2 \to 8 \to 2$	0.00563	Average relative error	7.50%
jmeint	Triangle intersection detection	3D gaming	10,000 random pairs of 3D triangle coordinates	32	0	23	1079	100,000 random pairs of 3D triangle coordinates	$18 \to 32 \to 8 \to 2$	0.00530	Miss rate	7.32%
jpeg	JPEG encoding	Compression	220 × 200-pixel color image	3	4	0	1257	Three 512 × 512-pixel color images	$64 \to 16 \to 64$	0.00890	Image difference	9.56%
kmeans	K-means clustering	Machine learning	220 × 200-pixel color image	1	0	0	26	50,000 pairs of random (*r*, *g*, *b*) values	$6 \to 8 \to 4 \to 1$	0.00169	Image difference	6.18%
sobel	Sobel edge detector	Image processing	220 × 200-pixel color image	3	2	1	88	One 512 × 512-pixel color image	$9 \to 8 \to 1$	0.00234	Image difference	3.44%

the Parrot transformation. For applications with random inputs, we use a different random input set. For applications with image input, we use a different image.

Code annotation. The C source code for each benchmark was annotated as described in Section 2: we identified a single pure function with fixed-size inputs and outputs. No algorithmic changes were made to the benchmarks to accommodate the Parrot transformation. There are many choices for the selection of target code and, for some programs, multiple NPUs may even have been beneficial. For the purposes of this evaluation, however, we selected a single target region per benchmark that was easy to identify, frequently executed as to allow for efficiency gains, and amenable to learning by a neural network. Qualitatively, we found it straightforward to identify a reasonable candidate function in each benchmark.

As shown in Table 1, in most of the benchmarks we examined, the target code contains complex control flow including conditionals, loops, and method calls. In jmeint, the target code contains the bulk of the algorithm, including many nested method calls and numerous conditionals. In jpeg, the transformation subsumes the discrete cosine transform and quantization phases, which contain function calls and loops. In fft, inversek2j, and sobel, the target code consists mainly of arithmetic operations and simpler control flow. In kmeans, the target code is the Euclidean distance calculation, which is simple and fine grained yet frequently executed. In each case, the target code is side-effect-free and the number of inputs/outputs is statically identifiable.

Training data. To train the NPU for each application, we have used either (1) typical program inputs (e.g., sample images) or (2) a limited number of random inputs. For the benchmarks that use random inputs, we determined the permissible range of parameters in the code and generated uniform random inputs in that range. For the image-based benchmarks, we used three standard images that are used to evaluate image processing algorithms. For kmeans, we supplied random inputs to the code region to avoid overtraining on a particular test image.

Output quality. We use an application-specific error metric to assess the QoR for each benchmark. In all cases, we compare the output of the original untransformed application to the output of the transformed application. For fft and inversek2j, which generate numeric outputs, we measure the average relative error. jmeint calculates whether two three-dimensional triangles intersect; we report the misclassification rate. For jpeg, kmeans, and sobel, which produce image outputs, we use the average root-mean-square image difference.

As the last column of Table 1 shows, application average error rates range from 3% to 10%. This QoR loss is commensurate with other work on quality trade-offs.[1, 8, 26]

[b] **Processor:** Fetch/Issue Width: 4/6, INT ALUs/FPUs: 3/2, Load/Store FUs: 2/2, ROB Entries: 96, Issue Queue Entries: 32, INT/FP Physical Registers: 256/256, Branch Predictor: Tournament 48KB, 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**: 32KB Instruction, 32KB Data, Line Width: 64 bytes, 8-Way, Latency: 3 cycles, **L2**: 2MB, Line Width 64 bytes, 8-Way, Latency: 12 cycles, **Memory Latency:** 50 ns (104 cycles).

6.2. Experimental setup

Simulation. We use the MARSSx86 cycle-accurate x86-64 simulator[22] to evaluate the performance effect of the Parrot transformation and NPU acceleration. We configure the simulator to resemble Intel's Penryn microarchitecture[b], which is an aggressive out-of-order design. We augment MARSSx86 with a cycle-accurate NPU simulator and add support for NPU queue instructions through unused x86 opcodes. We use C assembly inlining to add the NPU invocation code. We compile the benchmarks using GCC version 4.4.6 with the -O3 flag to enable aggressive compiler optimizations. The baseline in all of the reported results is the execution of the entire benchmark on the core without the Parrot transformation.

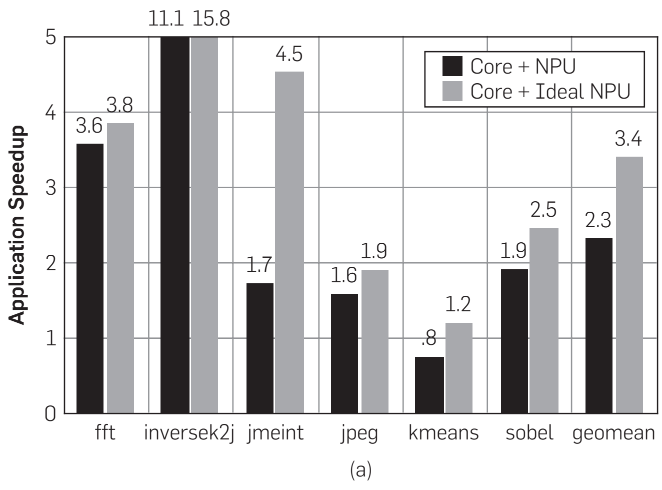
Energy modeling. MARSSx86 generates an event log during the cycle-accurate simulation of the program. The resulting statistics are sent to a modified version of McPAT[18] to estimate the energy consumption of each execution. We model the energy consumption of an 8-PE NPU using the results from McPAT and CACTI 6.5[20] for memory arrays, buses, and steering logic. We use the results from Galal and Horowitz[10] to estimate the energy of multiply-and-add operations. We model the NPU and the core at the 45 nm technology node. The NPU operates at the same frequency and voltage as the main core. We use the 2080 MHz frequency and $V_{dd} = 0.9$ V settings because the energy results in Galal and Horowitz[10] use this frequency and voltage setting.

6.3. Experimental results

Figure 7a shows the application speedup when an 8-PE NPU is used to replace each benchmark's target function. The rest of the code runs on the core. The baseline is executing the entire, untransformed benchmark on the CPU. The plots also show the potential available speedup: the hypothetical speedup if the NPU takes zero cycles for computation. Among the benchmarks, inversek2j sees the highest speedup (11.1×) since the Parrot transformation substitutes the bulk of the application with a relatively small NN ($2 \rightarrow 8 \rightarrow 2$). On the other hand, kmeans sees a 24% slowdown even though it shows a potential speedup of 20% in the limit. The transformed region of code in kmeans consists of 26 mostly arithmetic instructions that can efficiently run on the core while the NN ($6 \rightarrow 8 \rightarrow 4 \rightarrow 1$) for this benchmark is comparatively complex and involves more computation (84 multiply–adds and 12 sigmoids) than the original code. On average, the benchmarks see a speedup of 2.3× through NPU acceleration.

Figure 7b shows the energy reduction for each benchmark. The baseline is the energy consumed by running the entire benchmark on the unmodified CPU and the ideal energy savings for a hypothetical zero-energy NPU. The Parrot transformation elides the execution of significant portion of dynamic instructions that otherwise would go through power-hungry stages of the OoO pipeline. The reduction in the number of dynamic instructions and the energy-efficient design of the NPU yield a 3.0× average application energy reduction.

Figure 7. Performance and energy improvements: (a) total application speedup with 8-PE NPU and (b) total application energy saving with 8-PE NPU.



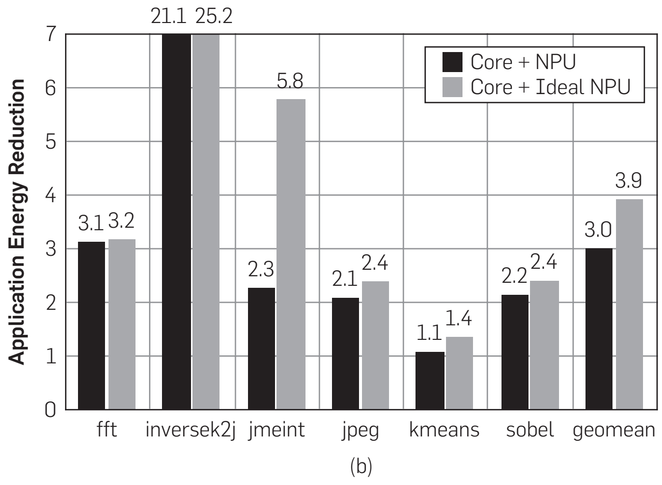
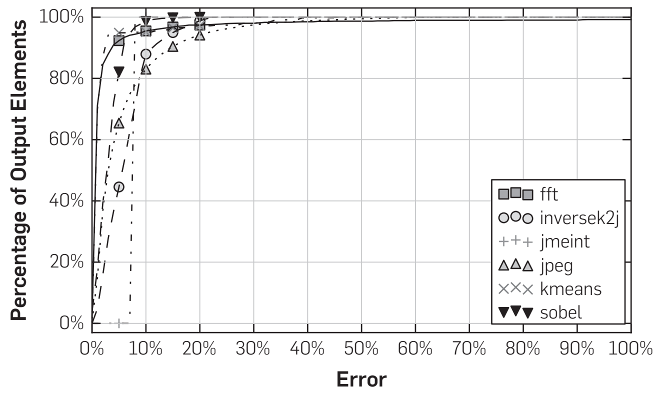


Figure 8. Cumulative distribution function (CDF) plot of the applications' output error. A point (*x*, *y*) indicates that *y* fraction of the output elements see error less than or equal to *x*.



To study the application-level quality loss in more detail, Figure 8 depicts the CDF (cumulative distribution function) plot of final error for each element of application's output. The output of each benchmark consists of a collection of elements—an image consists of pixels, a vector consists of scalars, etc. The error CDF reveals the distribution of output errors among an application's output elements and shows that very few output elements see large quality loss. The majority (80–100%) of each transformed application's output elements have error less than 10%.

7. RELATED WORK

This work represents a convergence of three main bodies of research: approximate computing, general-purpose configurable acceleration, and hardware neural networks. Fundamentally, the Parrot transformation leverages hardware neural networks to create a new class of configurable accelerators for approximate programs.

Approximate computing. Prior work has explored relaxed hardware semantics and their impact on applications with "soft" output requirements, both as (1) extensions to traditional architectures[4, 8, 19] and (2) in the form of fully approximate processing units.[2, 21] In contrast, NPUs accelerate coarse-grained blocks of code in larger applications. No special code must be written to take advantage of the approximate unit; only lightweight annotation is required. Some work has also exposed relaxed semantics in the programming language to give programmers control over the precision of software.[1, 4, 26] As an implementation of approximate semantics, the Parrot transformation dovetails with these programming models.

General-purpose configurable acceleration. The Parrot transformation extends prior work on configurable computing, synthesis, specialization, and acceleration that focuses on compiling traditional, imperative code for efficient hardware structures. One research direction seeks to synthesize efficient circuits or configure FPGAs to accelerate general-purpose code.[23, 24] Similarly, static specialization has shown significant efficiency gains for irregular and legacy code.[29] More recently, configurable accelerators have been proposed that allow the main CPU to offload certain code to a small, efficient structure.[11, 12] This work differs in its focus on accelerating *approximate* code. NPUs represent an opportunity to go beyond the efficiency gains that are possible when strict correctness is not required.

Neural networks. There is an extensive body of work on hardware implementation of neural networks (neural hardware) both digital[6] and analog.[17] Other work has examined the fault tolerance of hardware neural networks.[16, 28] A recent study[3] showed that 5 of 13 applications from the PARSEC suite can be manually reimplemented to make use of various kinds of neural networks.

8. LIMITATIONS AND FUTURE DIRECTIONS

Our results suggest that the Parrot transformation and NPU acceleration can provide significant performance and energy benefits. However, further research must address three limitations to the Parrot transformation as described in this work: (1) applicability, (2) programmer effort, and (3) quality and error control.

Applicability. Since neural networks inherently produce approximate results, not all code regions can undergo the

Parrot transformation. As enumerated in Section 3.1, a target code region must satisfy the following conditions:

- The region must be approximable. That is, the program must incorporate application-level tolerance of imprecision in the results of the candidate region.
- The region must have a bounded number of statically identifiable inputs and outputs.
- The region must be hot to benefit from acceleration.

Although these criteria form a basis for programmers or compilers to identify nominees for the Parrot transformation, they do not guarantee that the resulting neural network will accurately approximate the code region. There is no simple criterion that makes a certain task (here a candidate region) suited for learning by a neural network. However, our experience and results suggest that empirical assessment is effective to classify a wide variety of approximate functions as NPU-suitable. Follow-on work can improve on empirical assessment by identifying static code features that tend to indicate suitability for learning-based acceleration.

Programmer effort. In this paper, the Parrot transformation requires programmers to (1) identify approximable code regions and (2) provide application inputs to be used for training data collection.

As with the other approaches that ensure the safety of approximate computation and avoid catastrophic failures,[26] the programmer must explicitly provide information for the compiler to determine which code regions are safe to approximate. As Section 3.2 outlines, future work should explore allowing the compiler to automatically infer which blocks are amenable to approximation.

Because NPU acceleration depends on representative test cases, it resembles a large body of other techniques that use programmer-provided test inputs, including quality assurance (e.g., unit and integration testing) and profile-driven compilers. Future work should apply traditional coverage measurement and improvement techniques, such as test generation, to the Parrot transformation. In general, however, we found that it was straightforward to provide sufficient inputs for the programs we examined. This is in part because the candidate function is executed many times in a single application run, so a small number of inputs can suffice. Furthermore, as Section 4.2 mentions, an online version of the Parrot transformation workflow could use samples of postdeployment inputs if representative tests are not available predeployment.

Quality and error control. The results in this paper suggest that NPU acceleration can effectively approximate code with accuracy that is commensurate with state-of-the-art approximate computing techniques. However, there is always a possibility that, for some inputs, the NPU computes a significantly lower-quality result than the average case. In other words, without exhaustively exploring the NPU's input space, it is impossible to provide formal guarantees about its worst-case accuracy.

This unpredictability is common to other approximation techniques.[8, 26] As long as the frequency of low-quality results is low and the application can tolerate these infrequent large errors, approximation techniques like NPUs can be effective. For this reason, future research should explore mechanisms to mitigate the frequency of such low-quality results. One such mechanism is to predict whether the NPU execution of the candidate region will be acceptable. For example, one embodiment would check whether an input falls in the range of inputs seen previously during training. If the prediction is negative, the original code can be invoked instead of the NPU. Alternatively, the runtime system could occasionally measure the error by comparing the NPU output to the original function's output. In case the sampled error is greater than a threshold, the neural network can be retrained.

9. CONCLUSION

Traditionally, hardware implementations of neural networks have been confined to specific classes of learning applications due to lack of traditional programming models. In this paper, we showed that the potential exists to use them to mimic and accelerate general-purpose programs that can tolerate small errors. Our learning transformation provides the bridge between neural and von Neumann models of computing and enables a general-purpose use case for neural hardware. The acceleration capability of NPUs aligns with both transistor and application trends, as transistors become less reliable and as imprecise applications grow in importance. In fact, our work demonstrates that neural accelerators can successfully mimic diverse regions of approximable imperative code. The Parrot algorithmic transformation converts different regions of code to a common neural network representation. Using neural networks as the common representation enables a new class of accelerators, NPUs, that yield significant application-level energy and performance savings. The levels of error introduced are comparable to those seen in previous approximate computing techniques. Besides introducing the Parrot algorithmic transformation and a new class of accelerators, this work leads to the following two additional key insights. First, the program transformation must consider a range of neural network topologies; a single topology is ineffective across diverse applications. Second, the accelerator must be tightly coupled with a processor's pipeline to enable acceleration even when fine-grained regions of code are transformed. By providing an end-to-end solution to meet these key requirements, the evaluated application suite ran 2.3× faster on average while using 3.0× less energy and maintaining accuracy greater than 90% in all cases. NPUs form a new class of trainable accelerators with potential implementations in both the digital and analog domains.

Acknowledgments

We thank Brandon Lucia, Jacob Nelson, Ardavan Pedram, Renée St. Amant, Karin Strauss, Xi Yang, Amir Yazdanbakhsh, and the members of the Sampa group for their feedback. This work was supported in part by NSF grant CCF-1016495 and research support from Microsoft.

References

1. Baek, W. and Chilimbi, T.M. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI* (2010).
2. Chakrapani, L.N., Akgul, B.E.S., Cheemalavagu, S., Korkmaz, P., Palem, K.V., and Seshasayee, B. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE* (2006).
3. Chen, T., Chen, Y., Duranton, M., Guo, Q., Hashmi, A., Lipasti, M., Nere, A., Qiu, S., Sebag, M., Temam, O., and Bench, N.N. On the broad potential application scope of hardware neural network accelerators. In *IISWC* (2012).
4. de Kruijf, M., Nomura, S., and Sankaralingam, K. Relax: An architectural framework for software recovery of hardware faults. In *ISCA* (2010).
5. Dennard, R.H., Gaensslen, F.H., Rideout, V.L., Bassous, E., and LeBlanc, A.R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circ.* 9 (Oct. 1974), 256–268.
6. Esmaeilzadeh, H., Saeedi, P., Araabi, B.N., Lucas, C., and Fakhraie, S.M. Neural network stream processing core (NnSP) for embedded systems. In *ISCAS* (2006).
7. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Power challenges may end the multicore era. *Commun. ACM 56*, 2 (Feb. 2013), 93–102.
8. Esmaeilzadeh, H. et al. Architecture support for disciplined approximate programming. In *ASPLOS* (2012).
9. Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural acceleration for general-purpose approximate programs. In *MICRO* (2012).
10. Galal, S. and Horowitz, M. Energy-efficient floating-point unit design. *IEEE Trans. Comput. 60*, 7 (2011) 913–922.
11. Govindaraju, V., Ho, C.H., and Sankaralingam, K. Dynamically specialized datapaths for energy efficient computing. In *HPCA* (2011).
12. Gupta, S., Feng, S., Ansari, A., Mahlke, S., and August, D. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO* (2011).
13. Guzhva, A., Dolenko, S., and Persiantsev, I. Multifold acceleration of neural network computations using GPU. In *ICANN* (2009).
14. Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., and Horowitz, M. Understanding sources of inefficiency in general-purpose chips. In *ISCA* (2010).
15. Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. Toward dark silicon in servers. *IEEE Micro 31*, 4 (July–Aug. 2011), 6–15.
16. Hashmi, A., Berry, H., Temam, O., and Lipasti, M. Automatic abstraction and fault tolerance in cortical microarchitectures. In *ISCA* (2011).
17. Joubert, A., Belhadj, B., Temam, O., and Héliot, R. Hardware spiking neurons design: Analog or digital? In *IJCNN* (2012).
18. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., and Jouppi, N.P. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO* (2009).
19. Liu, S., Pattabiraman, K., Moscibroda, T., and Zorn, B.G. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS* (2011).
20. Muralimanohar, N., Balasubramonian, R., and Jouppi, N. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO* (2007).
21. Narayanan, S., Sartori, J., Kumar, R., and Jones, D.L. Scalable stochastic processors. In *DATE* (2010).
22. Patel, A., Afram, F., Chen, S., and Ghose, K. MARSx86: A full system simulator for x86 CPUs. In *DAC* (2011).
23. Putnam, A., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P., and Eggers, S. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *FPGA* (2008).
24. Razdan, R. and Smith, M.D. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO* (1994).
25. Rumelhart, D.E., Hinton, G.E., and Williams, R.J. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* D.E. Rumelhart, J.L. McClelland, and PDP Research Group, eds. Volume 1. MIT Press, 1986, 318–362.
26. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI* (2011).
27. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE* (2011).
28. Temam, O. A defect-tolerant accelerator for emerging high-performance applications. In *ISCA* (2012).
29. Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M.B. Conservation cores: Reducing the energy of mature computations. In *ASPLOS* (2010).
30. Zhu, J. and Sutton, P. FPGA implementations of neural networks: A survey of a decade of progress. In *FPL* (2003).

Hadi Esmaeilzadeh (hadi@cc.gatech.edu), Georgia Institute of Technology, Atlanta, GA.

Adrian Sampson and Luis Ceze ([asampson, luisceze]@cs.washington.edu), University of Washington, Seattle, WA.

Doug Burger (dburger@microsoft.com), Microsoft Research, Redmond, WA.