Appears in the Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012

Neural Acceleration for General-Purpose Approximate Programs

Hadi Esmaeilzadeh Adrian Sampson Luis Ceze Doug Burger*
University of Washington *Microsoft Research
{hadianeh,asampson,luisceze}@cs.washington.edu dburger@microsoft.com

Abstract

This paper describes a learning-based approach to the acceleration of approximate programs. We describe the Parrot transformation*, a program transformation that selects and trains a neural network to mimic a region of imperative code. 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 accelerate small code regions. Since neural networks produce inherently approximate results, we define a programming model that allows programmers to identify approximable code regions—code that can produce imprecise but acceptable results. 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 quality loss of at most 9.6%.*

1. Introduction

Energy efficiency is a primary concern in computer systems. The cessation of Dennard scaling has limited recent improvements in transistor speed and energy efficiency, resulting in slowed general-purpose processor improvements. Consequently, architectural innovation has become crucial to achieve performance and efficiency gains [10].

However, there is a well-known tension between efficiency and programmability. Recent work has quantified three orders of magnitude of difference in efficiency between general-purpose processors and ASICs [21, 36]. Since designing ASICs for the massive base of quickly changing, general-purpose applications is currently infeasible, practitioners are increasingly turning to programmable accelerators such as GPUs and FPGAs. Programmable accelerators provide an intermediate point between the efficiency of ASICs and the generality of conventional processors, gaining significant efficiency for restricted domains of applications.

Programmable accelerators exploit some characteristic of an application domain to achieve efficiency gains at the cost of generality. For instance, FPGAs exploit copious, fine-grained, and irregular parallelism but perform poorly when complex and frequent accesses to memory are required. GPUs exploit many threads and SIMD-style parallelism but lose efficiency when threads diverge. Emerging accelerators, such as BERET [19], Conservation Cores and QsCores [47, 48], or DySER [18], map regions of general-purpose code to specialized hardware units by leveraging either small, frequently-reused code idioms (BERET and DySER) or larger code regions amenable to hardware synthesis (Conservation Cores). 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 that is growing increasingly important. Many modern applications—such as image rendering, signal processing, augmented reality, data mining, robotics, and speech recognition—can tolerate inexact computation in substantial portions of their execution [7, 14, 28, 41]. This tolerance can be leveraged for substantial performance and energy gains.

This paper introduces a new class of programmable accelerators that exploit approximation for better performance and energy efficiency. The key idea is to *learn* how an original region of approximable code behaves and 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 [1, 8, 11, 29], or requires vastly different programming paradigms [4, 24, 26, 32]. Like emerging flexible accelerators [18, 19, 47, 48], our technique automatically offloads code segments from programs written in mainstream languages; but unlike prior work, it leverages changes in the semantics of the offloaded code.

We have identified three challenges that must be solved to realize effective trainable accelerators:

1. A **learning algorithm** is required that can accurately and efficiently mimic imperative code. We find that neural networks can approximate various regions of imperative code and propose the Parrot transformation, which exploits this finding (Section 2).
2. A **language and compilation framework** should be developed to transform regions of imperative code to neural network evaluations. To this end, we define a programming model and implement a compilation workflow to realize the Parrot transformation (Sections 3 and 4). The Parrot transformation starts from regions of approximable imperative code identified by the programmer, collects training data, explores the topology space of neural networks, trains them to mimic the regions, and finally replaces the original regions of code with trained neural networks.
3. An **architectural interface** is necessary to call a neural processing unit (NPU) in place of the original code regions. The NPU we designed is tightly integrated with a speculative out-of-order core. The low-overhead interface enables acceleration even when fine-grained regions of code are transformed. The core communicates both the neural configurations and run-time invocations to the NPU through extensions to the ISA (Sections 5 and 6).

Rather than contributing a new design for neural network implementation, this paper presents a new technique for harnessing hardware neural networks in general-purpose computations. We show that using neural networks to replace regions of imperative code is feasible and profitable by experimenting with a variety of applications, including FFT, gaming, clustering, and vision algorithms (Section 7). These applications do not belong to the class of modeling and prediction that typically use neural networks. For each application, we identify 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. Continuing work on NPU acceleration will provide a new class of accelerators—with implementation potential in both analog and digital domains—for emerging approximate applications.

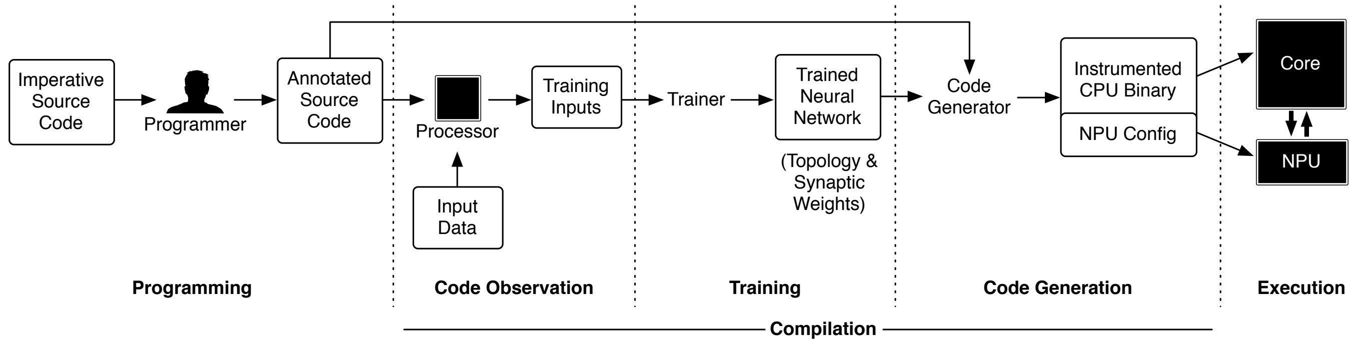


Figure 1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

2. Overview

The *Parrot transformation* is an algorithmic transformation that converts regions of imperative code to neural networks. Because neural networks expose considerable parallelism and can be efficiently accelerated using dedicated hardware, 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 neural processing unit (NPU), to perform neural evaluation instead of executing the replaced code. Figure 1 shows an overview of our proposed approach, 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. During development, the programmer explicitly annotates functions that are amenable to approximate execution and therefore candidates for the Parrot transformation. 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 is common practice in the approximate computing literature [8, 11, 41]. We discuss our programming model in detail in Section 3.

Compilation. Once the source code is annotated, as shown in Figure 1, the compiler applies the Parrot transformation in three steps: (1) code observation; (2) neural network selection and training; and (3) binary generation. Section 4 details these steps.

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 discover the topology of the neural network as well as its synaptic weights. It uses the backpropagation algorithm [40] coupled with a topology search (see Section 4.2) 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.

Many NPU implementations are feasible, from all-software 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 (Section 6) and architecture support to facilitate low-latency NPU invocations (Section 5).

A key insight in this paper is that it is possible to automatically discover and train neural networks that effectively approximate imperative code from diverse application domains. These diverse applications do not belong to the class of modeling and prediction applications that typically use neural networks. The Parrot transformation enables a novel use of hardware neural networks to accelerate many approximate applications.

3. Programming Model

The Parrot transformation starts with the programmer identifying candidate code regions. These candidate regions need to comply with certain criteria to be suitable for the transformation. 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

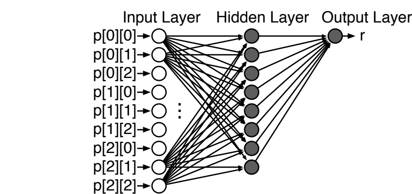
Candidate code for the Parrot transformation must satisfy three criteria: it must be frequently executed (i.e., a "hot" function); it must tolerate imprecision in its computation; and it must have well-defined inputs and outputs.

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

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

```
    void edgeDetection(Image& srcImg, Image& dstImg){
2     float[3][3] p; float pixel;
      for(int y = 0; y < srcImg.height; ++y)
4       for(int x = 0; x < srcImg.width; ++x)
          srcImg.toGrayeScale(x,y);
6     for(int y = 0; y < srcImg.height; ++y)
        for(int x = 0; x < scrImg.width; ++x){
8         p = srcImg.build3x3Window(x, y);
          pixel = sobel(p);
10        dstImg.setPixel(x, y, pixel);
        }
12  }
```

(a) Original implementation of the Sobel filter



(b) The sobel function transformed to a $9 \to 8 \to 1$ NN

```
    void edgeDetection(Image& srcImg, Image& dstImg){
2     float[3][3] p; float pixel;
      for(int y = 0; y < srcImg.height; ++y)
4       for(int x = 0; x < srcImg.width; ++x)
          srcImg.toGrayeScale(x,y);
6     for(int y = 0; y < srcImg.height; ++y)
        for(int x = 0; x < scrImg.width; ++x){
8         p = srcImg.build3x3Window(x, y);
          NPU_SEND(p[0][0]); NPU_SEND(p[0][1]); NPU_SEND(p[0][2]);
10        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]);
12        NPU_RECEIVE(pixel);
          dstImg.setPixel(x, y, pixel);
14      }
    }
```

(c) parrot transformed code; an NPU invocation replaces the function call

Figure 2: Three stages in the transformation of an edge detection algorithm using the Sobel filter.

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 2a 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 [2, 8, 29, 41, 43] 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 7). 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 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 2a) 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 be *pure:* it must not read any data other than its inputs nor affect any state other than its outputs (e.g., via a system call). These two criteria can be checked statically.

The sobel function in Figure 2a 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 [[annotation]] syntax as shown in Figure 2a). The programmer is responsible for ensuring that the function has no side effects, reads only its arguments, and writes only its return value. Each argument type and the return type must have a fixed size. 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.

Other annotation approaches. Our current system depends on explicit programmer annotations at the granularity of functions. While we find that explicit function annotations are straightforward to apply (see Section 7), static analysis techniques could be used to further simplify the annotation process. For example, in an approximation-aware programming language such as EnerJ [41], the programmer

uses type qualifiers to specify which data is non-critical and may be approximated. In such a system, the Parrot transformation can be automatically applied to any block of code that only affects approximate data. That is, the candidate regions for the Parrot transformation would be implicitly defined.

Like prior work on approximate computing, we acknowledge that some programmer guidance is essential when identifying error-tolerant code [2, 8, 11, 29, 41]. Tolerance to approximation is an inherently application-specific property. Fortunately, language-level techniques like EnerJ demonstrate that the necessary code annotations can be intuitive and straightforward for programmers to apply.

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

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 system also measures the minimum and maximum value for each input and output; the NPU normalizes values using these ranges during execution.

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. In our edge detection example, the sobel function runs for every pixel in the input image. So, as Section 7 details, training sobel on a single 512×512 test image provides 262144 training data points and results in acceptable accuracy when computing on unseen images.

Although we do not explore it in this paper, automatic input generation could help cover the space of possible inputs and thereby achieve a more accurate trained neural network. In particular, the compiler could synthesize new inputs by interpolating values between existing test cases.

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 [40] 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. The learning rate, a value between 0 and 1, is the step size of the gradient descent and identifies how much a single example affects the weights. Since backpropagation on MLPs is not convex and the compilation procedure is automatic, we choose a small learning rate of 0.01. Larger steps can cause oscillation in the training and prevent convergence. One complete pass over the training data is called an epoch. Since the learning rate is small, the epoch count should be large enough to ensure convergence. Empirically, we find that 5000 epochs achieve a good balance of generalization and accuracy. Larger epoch counts can cause overtraining and adversely affect the generalization ability of the network while smaller epoch counts may result in poor accuracy.

Neural network topology selection. In addition to running backpropagation, this phase selects a network topology that balances between 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 (see Figure 2b). 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.

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 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. (The numbers of neurons in the input and output layers are predetermined based on the number of inputs and outputs in the candidate function.) These choices limit the search space to 30 possible topologies. The maximum number of hidden layers and maximum neurons per hidden layer are compilation options and can be specified by the user. Although the candidate topologies can be trained in parallel, enlarging the search space increases the compilation time.

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. Figure 2b shows the three-layer MLP that replaces the sobel function. Each neuron in the network performs a weighted sum on its inputs and then applies a sigmoid function to the result of weighted sum.

On-line training. Our present system performs observation and training prior to deployment; an alternative design could train the neural network concurrently with in-vivo operation. On-line training could improve accuracy but would result in runtime overheads. To address these overheads, an on-line 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 via its configuration interface (Section 6.2). 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. The

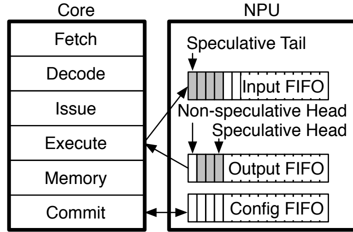


Figure 3: The NPU exposes three FIFO queues to the core. The speculative state of the FIFOs is shaded.

configuration and input–output communication occurs through ISA extensions discussed in Section 5.1.

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. Moreover, NPU invocation should not prevent speculative execution. This section discusses the ISA extensions and microarchitectural mechanism for tightly integrating the NPU with an out-of-order processor pipeline.

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. As shown in Figure 3, 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.

Instead of special instructions, an alternative design could use memory-mapped IO to communicate with the NPU. This design would require special fence instructions to prevent interference between two consecutive invocations and could impose a large overhead per NPU invocation.

5.2. Speculative NPU-Augmented Architecture

Scheduling and issue. To ensure correct communication with the NPU, the processor must issue NPU instructions in order. To accomplish this, the renaming logic implicitly considers every NPU instruction to read and write a designated "dummy" architectural register. The scheduler will therefore treat all NPU instructions as dependent. Furthermore, the scheduler only issues an enqueue instruction if the corresponding FIFO is not full. Similarly, a dequeue instruction is only issued if the corresponding FIFO is not empty.

Speculative execution. The processor can execute enq.d and deq.d instructions speculatively. Therefore, the head pointer of the input FIFO can only be updated—and consequently the entries recycled—when: (1) the enqueue instruction commits; and (2) the NPU finishes processing that input. When an enq.d instruction reaches the commit stage, a signal is sent to the NPU to notify it that the input FIFO head pointer can be updated.

To ensure correct speculative execution, the output FIFO maintains two head pointers: a speculative head and a non-speculative head. When a dequeue instruction is issued, it reads a value from the output FIFO and the speculative head is updated to point to the next output. However, the non-speculative head is not updated to ensure that the read value is preserved in case the issue of the instruction was a result of misspeculation. The non-speculative head pointer is only updated when the instruction commits, freeing the slot in the output FIFO.

In case of a flush due to branch or dependence misspeculation, the processor sends the number of squashed enq.d and deq.d instructions to the NPU. The NPU adjusts its input FIFO tail pointer and output FIFO speculative head pointer accordingly. The NPU also resets its internal control state if it was processing any of the invalidated inputs and adjusts the output FIFO tail pointer to invalidate any outputs that are based on the invalidated inputs. The rollback operations are performed concurrently for the input and output FIFOs.

The enq.c and deq.c instructions, which are only used to read and write the NPU configuration, are not executed speculatively.

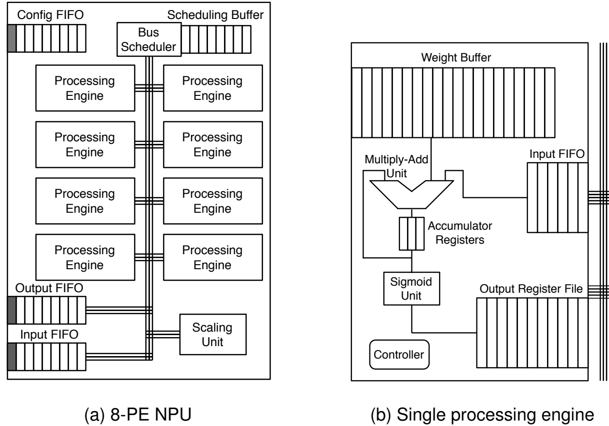
Interrupts. If an interrupt were to occur during an NPU invocation, the speculative state of the NPU would need to be flushed. The remaining non-speculative data in the input and output FIFOs would need to be saved and then restored when the process resumes. One way to avoid this complexity is to disable interrupts during NPU invocations; however, this approach requires that the invocation time is finite and ideally short as to not delay interrupts for too long.

Context switches. The NPU's configuration is architectural state, so the operating system must save and restore the configuration data on a context switch. The OS reads out the current NPU configuration using the deq.c instruction and stores it for later reconfiguration when the process is switched back in. To reduce context switch overheads, the OS can use the same lazy context switch techniques that are typically used with floating point units [33].

6. Neural Processing Unit

There are many implementation options for NPUs with varying tradeoffs in performance, power, area, and complexity, as illustrated by Figure 4. At one extreme are software implementations running on a CPU or GPU [20, 34]. Since these implementations have higher computation and communication overheads, they are likely more suitable for very large candidate regions, when the invocation cost

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



(a) 8-PE NPU (b) Single processing engine

Figure 5: Reconfigurable 8-PE NPU.

can be better amortized. Next on the scale are FPGA-based implementations [50]. Digital ASIC designs are likely to be lower-latency and more power-efficient than FPGA-based implementations [9, 37]. Since neural networks are themselves approximable, their implementation can also be approximate. Therefore, we can improve efficiency further and use approximate digital circuits (e.g., sub-critical voltage supply). In the extreme, one can even use custom analog circuitry or FPAAs [3, 42, 44]. In fact, we believe that analog NPUs have significant potential and we plan to explore them in future work. We focus on an ASIC design operating at the same critical voltage as the main core. This implementation represents a reasonable trade-off between efficiency and complexity; it is able to accelerate a wide variety of applications without the complexity of integrating analog or sub-critical components with the processor.

6.1. Reconfigurable Digital NPU

The Parrot transformation produces different neural network topologies for different code regions. Thus, we propose a reconfigurable NPU design that accelerates the evaluation of a range of neural topologies. As shown in Figure 5a, the NPU contains eight identical processing engines (PEs) and one scaling unit. Although the design can scale to larger numbers of PEs, we find that the speedup gain beyond 8 PEs is small (see Section 7). 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.

The NPU stores the bus scheduling information in its circular scheduling buffer (shown in Figure 5a). Each entry in this buffer schedules the bus to send a value from a PE or the input FIFO to a set of destination PEs or the output FIFO. Every scheduling buffer entry consists of a source and a destination. The source is either the input FIFO or the identifier of a PE along with an index into its output register file (shown in Figure 5b). The destination is either the output FIFO or a bit field indicating the destination PEs.

Figure 5b 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 multilayer perceptron, each neuron computes its output as $y = \text{sigmoid}(\sum_i (x_i \times w_i))$ 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.

Each entry in the weight buffer is augmented by one bit indicating whether a neuron's multiply-add operation has finished. When it finishes, the PE applies the sigmoid function, which is implemented as a lookup table, and write the result to its output register file. The per-neuron information stored in the weight buffer also indicates which output register should be used.

6.2. NPU Configuration

During code generation (Section 4.3), the compiler produces an NPU configuration that implements the trained neural network for each candidate function. The static NPU scheduling algorithm first assigns an order to the inputs of the neural network. This order determines both the sequence of enq.d instructions that the CPU will send to the NPU during each invocation and the order of multiply-add operations among the NPU's PEs. Then, the scheduler takes the following steps for each layer of the neural network:

1. Assign each neuron to one of the processing engines.
2. Assign an order to the multiply-add operations considering the order assigned to the inputs of the layer.
3. Assign an order to the outputs of the layer.
4. Produce a bus schedule reflecting the order of operations.

The ordering assigned for the final layer of the neural network dictates the order in which the program will retrieve the NPU's output using deq.d instructions.

7. 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 perform a sensitivity analysis to examine the effect of NPU PE count and communication latency on the performance benefits.

7.1. Benchmarks and the Parrot Transformation

Table 1 lists the benchmarks used in this evaluation. 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 [1, 11, 28, 29, 41, 43].

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 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.

Table 1: The benchmarks evaluated, characterization of each transformed function, 0 data, and the result of the Parrot transformation.

	Description	Type	Evaluation Input Set	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 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	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	Average Relative Error	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	10000 (x,y) Random Coordinates	4	0	0	100	10000 (x,y) Random Coordinates	2 -> 8 -> 2	0.00563	Average Relative Error	7.50%
jmeint	Triangle intersection detection	3D Gaming	10000 Random Pairs of 3D Triangle Coordinates	32	0	23	1,079	100000 Random Pairs of 3D Triangle Coordinates	18 -> 32 -> 8 -> 2	0.00530	Miss Rate	7.32%
jpeg	JPEG encoding	Compression	220x200-Pixel Color Image	3	4	0	1,257	Three 512x512-Pixel Color Images	64 -> 16 -> 64	0.00890	Image Diff	9.56%
kmeans	K-means clustering	Machine Learning	220x200-Pixel Color Image	1	0	0	26	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00169	Image Diff	6.18%
sobel	Sobel edge detector	Image Processing	220x200-Pixel Color Image	3	2	1	88	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.00234	Image Diff	3.44%

Code annotation. The C source code for each benchmark was annotated as described in Section 3: 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.

Table 1 shows the number of function calls, conditionals, and loops in each transformed function. The table also shows the number of x86-64 instructions for the target function when compiled by GCC 4.4.6 at the -O3 optimization level. We do not include the statistics of the standard library functions in these numbers. In most of these benchmarks, 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 0 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 are 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 (lena, mandrill, and peppers). For kmeans, we supplied random inputs to the code region to avoid overtraining on a particular test image. Table 1 shows the specific image or application input used in the training phase for each benchmark. We used different random inputs and different images for the final accuracy evaluation.

Neural networks. The "Neural Network Topology" column in Table 1 shows the topology of the trained neural network discovered by the training stage described in Section 4.2. The "NN MSE" column shows the mean squared error for each neural network on the test subset of the training data. For example, the topology for jmeint is $18 \rightarrow 32 \rightarrow 8 \rightarrow 2$, meaning that the neural network takes in 18 inputs, produces 2 outputs, and has two hidden layers with 32 and 8 neurons respectively. As the results show, the compilation workflow was able to find a neural network that accurately mimics each original function. However, different topologies are required to approximate different functions.

> Different applications require different neural network topologies, so the NPU structure must be reconfigurable.

Output quality. We use an application-specific error metric, shown in Table 1, to assess the quality of each benchmark's output. 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. The column labeled "Error" in Table 1 shows the whole-application error of each benchmark according to its error metric. Unlike the "NN MSE" error values, this application-level error assessment accounts for accumulated errors due to repeated execution of the transformed function.

Application average error rates range from 3% to 10%. This quality-of-service loss is commensurate with other work on quality trade-offs. Among hardware approximation techniques, Truffle [11] shows similar error (3–10%) for some applications and much greater error (above 80%) for others in a moderate configuration. The evaluation of EnerJ [41] also has similar error rates; two thirds of the applications exhibit error greater than 10% in the most energy-efficient configuration. Green [2], a software technique, has error rates below 1% for some applications but greater than 20% for others. A case study by Misailovic et al. [30] explores manual optimizations of a video encoder, x264, that trade off 0.5–10% quality loss.

Table 2: Microarchitectural parameters for the core, caches, memory, NPU, and each PE in the NPU.

Core		Caches and Memory		NPU	
Architecture	x86-64	L1 Cache Size	32 KB instruction, 32 KB data	Number of PEs	8
Fetch/Issue Width	4/6	L1 Line Width	64 bytes	Bus Schedule FIFO	512×20-bit
INT ALUs/FPUs	3/2	L1 Associativity	8	Input FIFO	128×32-bit
Load/Store FUs	2/2	L1 Hit Latency	3 cycles	Output FIFO	128×32-bit
ROB Entries	96	ITLB/DTLB Entries	128/256	Config FIFO	8×32-bit
Issue Queue Entries	32	L2 Cache Size	2 MB	**NPU PE**	
INT/FP Physical Registers	256/256	L2 Line Width	64 bytes		
Branch Predictor	Tournament, 48 KB	L2 Associativity	8	Weight Cache	512×33-bit
BTB Sets/Ways	1024/4	L2 Hit Latency	12	Input FIFO	8×32-bit
RAS Entries	64	Memory Latency	50 ns (104 cycles)	Output Register File	8×32-bit
Load/Store Queue Entries	48/48			Sigmoid Unit LUT	2048×32-bit
Dependence Predictor	4096-entry Bloom Filter			Multiply-Add Unit	32-bit Single-Precision FP

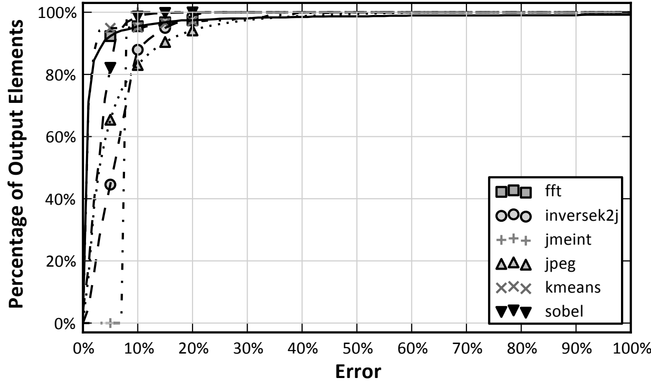


Figure 6: Cumulative distribution function (CDF) plot of the applications' output error. A point (x,y) indicates that y fraction of the output elements see error less than or equal to x.

> The Parrot transformation degrades each application's average output quality by less than 10%, a rate commensurate with other approximate computing techniques.

To study the application level quality loss in more detail, Figure 6 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% to 100%) of each transformed application's output elements have error less than 10%.

7.2. Experimental Setup

Cycle-accurate simulation. We use the MARSSx86 cycle-accurate x86-64 simulator [35] to evaluate the performance effect of the Parrot transformation and NPU acceleration. Table 2 summarizes the microarchitectural parameters for the core, memory subsystem, and NPU. We configure the simulator to resemble Intel's Penryn microarchitecture, 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 [27] 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 [31] for memory arrays, buses, and steering logic. We use the results from Galal et al. [17] 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 et al. [17] are for this frequency and voltage setting.

7.3. Experimental Results

Dynamic instruction subsumption. Figure 7 depicts dynamic instruction count of each transformed benchmark normalized to the instruction count for CPU-only execution. The figure divides each application into NPU communication instructions and application instructions. While the potential benefit of NPU acceleration is directly related to the amount of CPU work that can be elided, the queuing instructions and the cost of neural network evaluation limit the actual benefit. For example, inversek2j exhibits the greatest potential for benefit: even accounting for the communication instructions, the transformed program executes 94% fewer instructions on the core. Most of the benchmark's dynamic instructions are in the target region for the Parrot transformation and it only communicates four values with the NPU per invocation. This is because inversek2j is an ideal case: the entire algorithm has a fixed-size input ((x,y) coordinates of the robot arm), fixed-size output ((θ_1, θ_2) angles for the arm joints), and tolerance for imprecision. In contrast, kmeans is representative of applications where the Parrot transformation applies more locally: the target code is "hot" but only consists of a few arithmetic operations and the communication overhead is relatively high.

Performance and energy benefits. Figure 8a 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

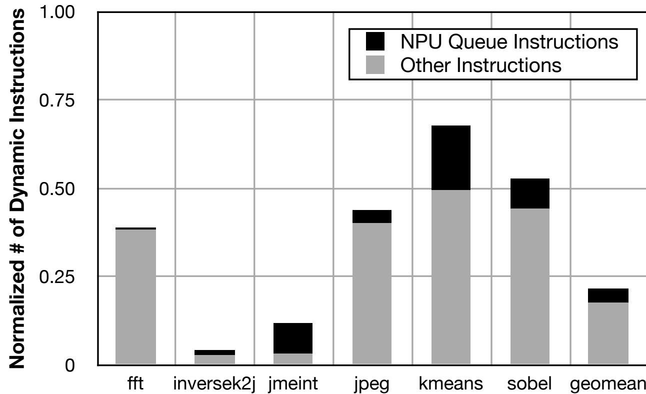
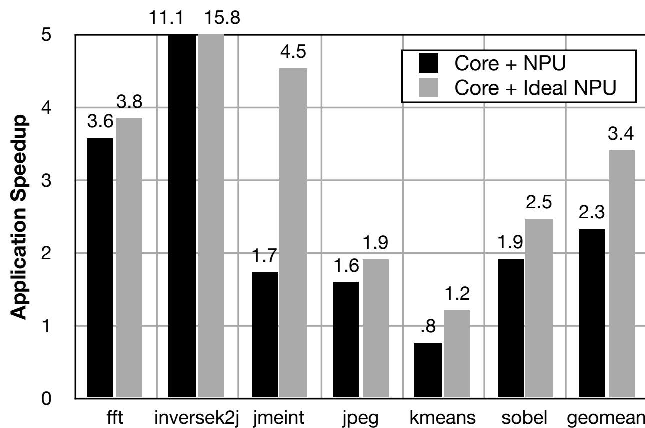
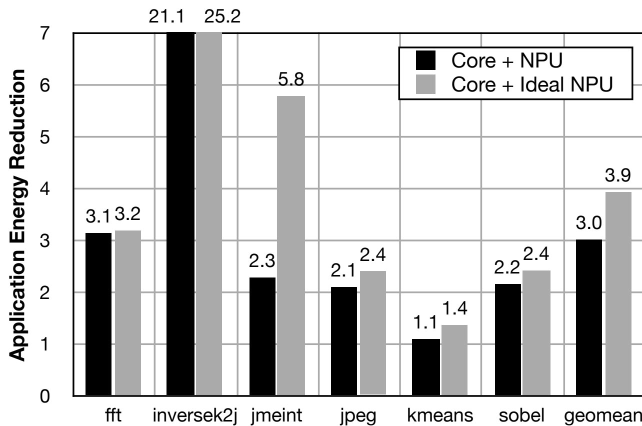


Figure 7: Number of dynamic instructions after Parrot transformation normalized to the original program.



(a) Total application speedup with 8-PE NPU



(b) Total application energy saving with 8-PE NPU

Figure 8: Performance and energy improvements.

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 8b 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.

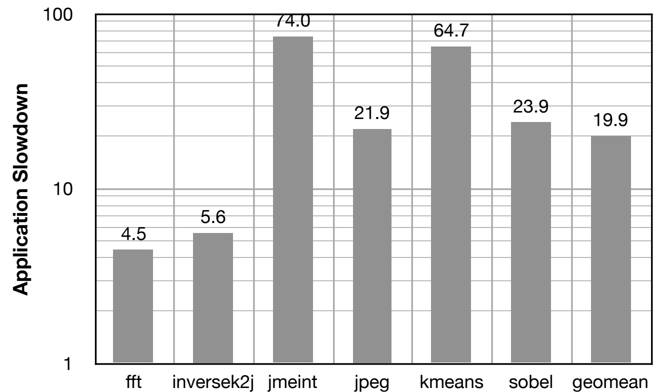


Figure 9: Slowdown with software neural network execution.

> For the applications we studied, the Parrot transformation and NPU acceleration provided an average 2.3× speedup and 3.0× energy reduction.

> Results for a hypothetical zero-cost NPU suggest that, in the limit, more efficient implementation techniques such as analog NPUs could result in up to 3.4× performance and 3.7× energy improvements on average.

Software neural network execution. While our design evaluates neural networks on a dedicated hardware unit, it is also possible to run transformed programs entirely on the CPU using a software library for neural network evaluation. To evaluate the performance of this all-software configuration, we executed each transformed benchmark using calls to the widely-used Fast Artificial Neural Network (FANN) library [15] in place of NPU invocations. Figure 9 shows the slowdown compared to the baseline (untransformed) execution of each benchmark. Every benchmark exhibits a significant slowdown when the Parrot transformation is used without NPU acceleration. jmeint shows the highest slowdown because 1079 x86 instructions—which take an average of 326 cycles on the core—are replaced by 928 multiplies, 928 adds, and 42 sigmoids. FANN's software multiply-and-add operations involve calculating the address of the neuron weights and loading them. The overhead of function calls in the FANN library also contributes to the slowdown.

> The Parrot transformation requires efficient neural network execution, such as hardware acceleration, to be beneficial.

Sensitivity to communication latency. The benefit of NPU-based execution depends on the cost of each NPU invocation. Specifically, the latency of the interconnect between the core and the NPU can affect the potential energy savings and speedup. Figure 10 shows the speedup for each benchmark under five different communication latencies. In each configuration, it takes n cycles to send data to the NPU and n cycles to receive data back from the NPU. In effect, $2n$ cycles are added to the NPU invocation latency. We imagine a design with pipelined communication, so individual enqueue and dequeue instructions take one extra cycle each in every configuration.

The effect of communication latency varies depending on the application. In cases like jpeg, where the NPU computation latency is significantly larger than the communication latency, the speedup is mostly unaffected by increased latency. In contrast, inversek2j sees

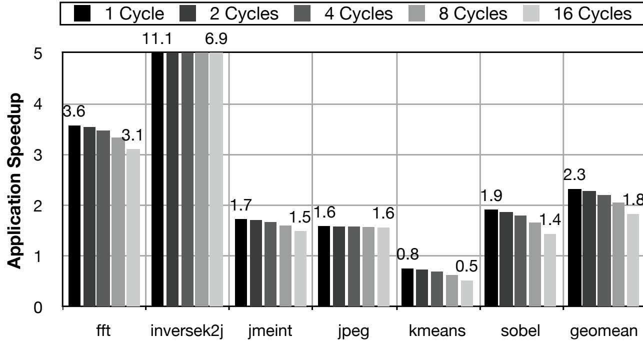


Figure 10: Sensitivity of the application's speedup to NPU communication latency. Each bar shows the speedup if communicating with the NPU takes n cycles.

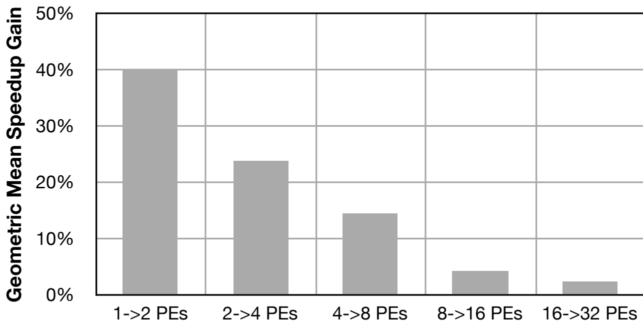


Figure 11: Performance gain per doubling the number of PEs.

a significant reduction in speedup from 11.1× to 6.9× when the communication latency increases from one cycle to 16 and becomes comparable to the computation latency. For kmeans, the slowdown becomes 48% for a latency of 16 cycles compared to 24% when the communication latency is one cycle.

> For some applications with simple neural network topologies, a tightly-coupled, low-latency NPU–CPU integration design is highly beneficial. Other applications we studied can tolerate a higher-latency interconnect.

Number of PEs. Figure 11 shows the geometric mean speedup gain from doubling the number of PEs in the NPU. Doubling the number of PEs beyond eight yields less than 5% geometric mean speedup gain, which does not justify the complexity of adding more than eight PEs for our benchmarks.

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 hot in order to benefit from acceleration.
- 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.

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 [41], 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 on-line version of the Parrot transformation workflow could use samples of post-deployment inputs if representative tests are not available pre-deployment.

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 give guarantees about its worst-case accuracy.

This unpredictability is common to other approximation techniques [11,41]. 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. These techniques are similar in spirit to related research on estimating error bounds for neural networks [46].

9. 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. Many categories of "soft" applications have been shown to be tolerant to imprecision during execution [7, 14, 28, 49]. Prior work has explored relaxed hardware semantics and their impact on these applications, both as (1) extensions to traditional architectures [1, 8, 11, 29] and (2) in the form of fully approximate processing units [4, 24, 26, 32].

In the former category (1), a conventional processor architecture is extended to enable selective approximate execution. Since all the instructions, both approximate and precise, still run on the core, the benefits of approximation are limited. In addition, these techniques' fine granularity precludes higher-level, algorithmic transformations that take advantage of approximation. The Parrot transformation operates at coarser granularities—from small functions to entire algorithms—and potentially increases the benefits of approximation. Furthermore, NPU acceleration reduces the number of instructions that go through the power-hungry frontend stages of the processor pipeline. In the latter category (2), entire processing units carry relaxed semantics and thus require vastly different programming models. In contrast, NPUs can be used with conventional imperative 0 languages and existing code. 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 [2, 8, 41]. 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 [6, 13, 38, 39]. Similarly, static specialization has shown significant efficiency gains for irregular and legacy code [47, 48]. More recently, configurable accelerators have been proposed that allow the main CPU to offload certain code to a small, efficient structure [18, 19]. These techniques, like NPU acceleration, typically rely on profiling to identify frequently executed code sections and include compilation workflows that offload this "hot" code to the accelerator. 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. While some code is not amenable to approximation and should be accelerated only with correctness-preserving techniques, NPUs can provide greater performance and energy improvements in many situations where relaxed semantics are appropriate.

Neural networks. There is an extensive body of work on hardware implementation of neural networks (neural hardware) both digital [9, 37, 50] and analog [3, 25, 42, 44]. Recent work has proposed higher-level abstractions for implementation of neural networks [23]. Other work has examined fault-tolerant hardware neural networks [22, 45]. In particular, Temam [45] uses datasets from the UCI machine learning repository [16] to explore fault tolerance of a hardware neural network design. That work suggests that even faulty hardware can be used for efficient simulation of neural networks. The Parrot algorithmic transformation provides a compiler workflow that allows general-purpose approximate applications to take advantage of this and other hardware neural networks.

An early version of this work [12] proposed the core idea of automatically mapping approximable regions of imperative code to neural networks. A more recent study [5] showed that 5 of 13 applications from the PARSEC suite can be manually reimplemented to make use of various kinds of neural networks, demonstrating that some applications allow higher-level algorithmic modifications to make use of hardware neural networks (and potentially an architecture like NPUs). However, that work did not prescribe a programming model nor a preferred hardware architecture.

10. Conclusion

This paper demonstrates that neural accelerators can successfully mimic diverse regions of approximable imperative code. Using this neural transformation and the appropriate hardware accelerator, significant application-level energy and performance savings are achievable. The levels of error introduced are comparable to those seen in previous approximate computing techniques. For the technique to be effective, two challenges must be met. 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 accelerate fine-grained regions of code. With these requirements met, our application suite ran 2.3× faster on average while using 3.0× less energy and maintaining accuracy greater than 90% in all cases.

Traditionally, hardware implementations of neural networks have been confined to specific classes of learning applications. In this paper, we show that the potential exists to use them to accelerate general-purpose code that can tolerate small errors. In fact, the transformation was successful for every approximable code region that we tested. This acceleration capability aligns with both transistor and application trends, as transistors become less reliable and as imprecise applications grow in importance. NPUs may thus form a new class of trainable accelerators with potential implementations in the digital and analog domains.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We thank our shepherd, Mike Schlansker, for his feedback and encouragement. We also thank Brandon Lucia, Jacob Nelson, Ardavan Pedram, Renée St. Amant, Karin Strauss, Xi Yang, and the members of the Sampa group for their feedback on the manuscript. This work was supported in part by NSF grant CCF-1016495 and gifts from Microsoft.

References

[1] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, 2005.

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

[3] B. E. Boser, E. Säckinger, J. Bromley, Y. Lecun, L. D. Jackel, and S. Member, "An analog neural network processor with programmable topology," *J. Solid-State Circuits*, vol. 26, pp. 2017–2025, 1991.

[4] 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.

[5] 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*, Nov. 2012.

[6] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[7] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in *SELSE*, 2009.

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

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

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[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, "Towards neural acceleration for general-purpose approximate computing," in *WEED*, Jun. 2012.

[13] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *HPCA*, 2009.

[14] Y. Fang, H. Li, and X. Li, "A fault criticality evaluation framework of digital systems for error tolerant video applications," in *ATS*, 2011.

[15] FANN, "Fast artificial neural network library," 2012. Available: http://leenissen.dk/fann/wp/

[16] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. Available: http://archive.ics.uci.edu/ml

[17] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 913–922, 2011.

[18] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011.

[19] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[20] A. Guzhva, S. Dolenko, and I. Persiantsev, "Multifold acceleration of neural network computations using GPU," in *ICANN*, 2009.

[21] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ISCA*, 2010.

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

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

[24] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED*, 1999.

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

[26] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.

[27] 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.

[28] X. Li and D. Yeung, "Exploiting soft computing for increased fault tolerance," in *ASGI*, 2006.

[29] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *ASPLOS*, 2011.

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

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

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

[33] NetBSD Documentation, "How lazy FPU context switch works," 2011. Available: http://www.netbsd.org/docs/kernel/lazyfpu.html

[34] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, 2004.

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

[36] A. Pedram, R. A. van de Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *Computers, IEEE Transactions on*, vol. 61, no. 12, Dec. 2012.

[37] K. Przytula and V. P. Kumar, Eds., *Parallel Digital Implementations of Neural Networks*. Prentice Hall, 1993.

[38] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *FPGA*, 2008.

[39] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *MICRO*, 1994.

[40] 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, pp. 318–362.

[41] 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.

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

[43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.

[44] S. Tam, B. Gupta, H. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in *SMC*, 1990.

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

[46] N. Townsend and L. Tarassenko, "Estimations of error bounds for neural-network function approximators," *IEEE Transactions on Neural Networks*, vol. 10, no. 2, Mar. 1999.

[47] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.

[48] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, S. Swanson, and M. Taylor, "QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MICRO*, 2011.

[49] V. Wong and M. Horowitz, "Soft error resilience of probabilistic inference applications," in *SELSE*, 2006.

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