Neural Acceleration for General-Purpose Approximate Programs

This work proposes an approximate algorithmic transformation and a new class of accelerators, called neural processing units (NPUs). NPUs leverage the approximate algorithmic transformation that converts regions of code from a von Neumann model to a neural model. NPUs achieve an average 2.3× speedup and 3.0× energy savings for general-purpose approximate programs. This new class of accelerators shows that significant performance and efficiency gains are possible when the abstraction of full accuracy is relaxed in general-purpose computing.

Hadi Esmaeilzadeh
Adrian Sampson
Luis Ceze
University of Washington
Doug Burger
Microsoft Research

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.[1]

However, a tension exists between efficiency and programmability. Recent work has quantified three orders of magnitude of difference in efficiency between general-purpose processors and application-specific integrated circuits.[2] 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 field-programmable gate arrays (FPGAs). Programmable accelerators provide an intermediate point between the efficiency of ASICs and the generality of conventional processors, gaining significant efficiency for restricted application domains.

Programmable accelerators exploit some characteristic of an application domain to gain efficiency at the cost of generality. FPGAs, for example, exploit copious, fine-grained, and irregular parallelism, while GPUs exploit many threads and 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 that is growing increasingly important. Many modern applications—such as image rendering, signal processing, augmented reality, data analytics, robotics, and speech recognition—can tolerate inexact computation in substantial portions of their execution.[3-6] This tolerance can be leveraged for substantial performance and energy gains.

This article introduces a new class of programmable accelerators that exploit approximation for better performance and

Published by the IEEE Computer Society

energy efficiency. The key idea is to learn how an original region of approximable code behaves and to 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,[7-9] or requires vastly different programming paradigms.[10-11] As with emerging flexible accelerators,[12-14] our technique automatically offloads code segments from programs written in mainstream languages; but unlike prior work, it leverages changes in the offloaded code's semantics.

Many specialized accelerators are limited to a single class of workloads or require significant programmer effort to transform the workload into a form that the accelerator can use. The key to our approach is that the algorithmic transformation converts diverse approximable regions of code into a common representation that can be efficiently executed on an accelerator. This work shows that using neural networks as the common representation can lead to significant performance and efficiency gains because neural networks consist of simple, regular, parallel operations. Unlike the CPU, a neural processing unit takes advantage of hard-wired control and avoids the overhead of fetching and decoding instructions. This work's main contribution is the automatic transformation from code regions to a common neural-network representation. This transformation structurally and semantically changes the code and replaces various unstructured regions of code with structured fine-grained parallel computation that is the neural network. The transformation is possible because the abstraction of full accuracy is relaxed in the computation. The idea of learning regions of code is another contribution of this work.

Approximate computing

Relaxing the high tax of providing perfect accuracy at the device, circuit, architecture, and programming language levels can provide significant opportunities to improve performance and energy efficiency for the domains in which applications can tolerate approximation.[8-11,15,16] These applications span embedded systems that operate on sensory inputs to multimedia, vision, web search, machine learning, data analytics, optimization, and more. While conventional techniques—such as dynamic voltage and frequency scaling—trade performance for energy, approximate computing trades error for performance and energy gains. Four broad categories of applications can benefit from general-purpose approximate computing:

- applications with analog inputs, such as sensory data processing and scene reconstructions in augmented reality;
- applications with analog output, such as multimedia;
- applications with multiple acceptable answers, such as web search and machine learning; and
- convergent applications, such as data analytics and optimization.

These diverse classes of applications provide opportunities for general-purpose approximate computing on both mobile and server computing systems.

However, applying approximation without discipline would make it nearly impossible to construct reliable software and could lead to catastrophic failures during execution. For approximate computation to be safe, it must be confined to the error-tolerant parts of the program. It must not, for example, lead to uncontrolled jumps or wild pointers. This need for a balance between approximate and traditional execution has led to research on disciplined approximation approaches. At the programming-language level, we introduced the EnerJ approximation-aware language, which lets programmers safely distinguish error-tolerant program components and protect critical components from errors.[3] At the architecture level, we introduced a variable-precision instruction set architecture (ISA) that allows conventional Von Neumann processors to interleave approximate and precise operations at a single-instruction granularity.[9] This ISA allows the compiler to convey what can be approximated without specifying how, letting the microarchitecture choose from a

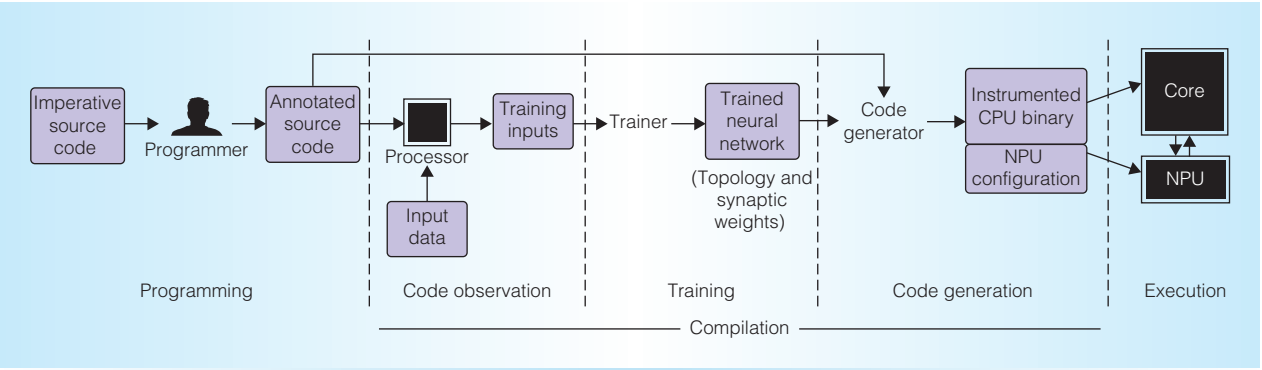

Figure 1. The Parrot transformation at a glance: from annotated code to accelerated execution on a neural-processing-unit (NPU) augmented core. The Parrot transformation has three key phases: programming, in which the programmer marks code regions to be transformed; compilation, in which the compiler selects and trains a suitable neural network and replaces the original code with a neural network invocation; and execution.

range of approximation techniques without exposing them to software. We also designed the dual-voltage Truffle microarchitecture, which implements this variable-precision ISA.

Although simulation results for that architecture showed energy savings up to 43 percent, the traditional processor design's constraints limit the efficiency gains for this approach. We can only optimize part of the Von Neumann pipeline with this approach—the front end, including instruction decode and control, can't be approximated. However, this article introduces an algorithmic transformation that converts an approximable code region from a Von Neumann model to a neural model, enabling much larger performance and efficiency gains.

Overview

The Parrot transformation is an algorithmic transformation that converts regions of imperative code into 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 neural processing unit (NPU), to perform neural evaluation instead of executing the replaced code. Figure 1 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.

For information on work related to the Parrot transformation and NPU acceleration, see the "Research Related to the Parrot Transformation and Neural Acceleration" sidebar.

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 application's overall reliability. This is common practice in the approximate computing literature.[3,8,9]

Specifically, to identify candidate functions, the programmer marks them as approximable with an annotation (for example, using C++11 [[annotation]] syntax) that satisfies the following requirements:

- *Hot code.* To benefit from acceleration, the offloaded code must account for a large portion of the program's work.

- *Approximable.* Because neural networks are inherently approximate, the program must be able to tolerate imprecision in the target code's output. Previous work has shown that programmers can identify such soft code in approximate applications.[3,8,9]
- *Well-defined inputs and outputs.* To efficiently use a neural network, the code must have fixed-size, statically identifiable inputs and outputs. It must not, for example, allocate a dynamically sized array as its output.

After the programmer identifies suitable functions, the transformation that enables NPU acceleration is completely automatic.

Compilation

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

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 back-propagation algorithm 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 executing 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 whole program is accelerated.

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's 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.

A key insight in this article is that it's possible to automatically discover and train neural networks that effectively approximate imperative code from diverse application domains. These diverse applications don't belong to the class of modeling and prediction applications that typically use neural networks. As Figure 2a illustrates, the Parrot transformation converts diverse regions of code to a common representation—neural networks. Using neural networks as a common representation during compilation enables a novel use of hardware neural networks to accelerate many approximate applications. As the results from this work suggest, different applications require different neural topologies. Thus, we design one reconfigurable digital NPU to accelerate many applications.

Compilation workflow

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

Research Related to the Parrot Transformation and Neural Acceleration

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 application are tolerant to imprecision during execution. Prior work has explored relaxed hardware semantics and their impact on these applications, both as extensions to traditional architectures[1-4] and in the form of fully approximate processing units.[5-8] In contrast, NPUs accelerate coarse-grained blocks of code in larger applications. No special code must be written to take advantage of approximate units of processing; only lightweight annotation is required.

Some work has also exposed relaxed semantics in programming languages to give programmers control over software precision.[2,9,10] 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.[11,12] Similarly, static specialization has shown significant efficiency gains for irregular and legacy code.[13] More recently, researchers have proposed configurable accelerators that let the main CPU offload certain code to a small, efficient structure.[14,15] 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

Researchers have extensively studied hardware implementations of neural networks (neural hardware), both digital[16] and analog.[17] Recent work has proposed higher-level abstractions for neural-network implementation.[18] Other work has examined fault-tolerant hardware neural networks.[19] In particular, Temam uses datasets from the UCI machine learning repository to explore a hardware neural network design's fault tolerance.[20] 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 exploit this and other hardware neural networks.

A recent study showed that five of 13 applications from the PARSEC (Princeton Application Repository for Shared-Memory Computers) suite can be manually reimplemented to use various kinds of neural networks, demonstrating that some applications allow higher-level algorithmic modifications to use hardware neural networks (and potentially an architecture like NPUs).[19] However, that work did not propose an algorithmic transformation and did not prescribe a programming model or preferred hardware architecture.

References

1. C. Alvarez et al., ''Fuzzy Memoization for Floating-Point Multimedia Applications,'' *IEEE Trans. Computers,* July 2005, pp. 922-927.
2. M. de Kruijf, S. Nomura, and K. Sankaralingam, ''Relax: An Architectural Framework for Software Recovery of Hardware Faults,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), 2010, pp. 497-508.
3. H. Esmaeilzadeh et al., ''Architecture Support for Disciplined Approximate Programming,'' *Proc. 17th Int'l Conf. Architectural*

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 annotated function's input and output. 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 might 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.

Training

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

The compiler uses the back-propagation algorithm to train the neural network.

Support for Programming Languages and Operating Systems, ACM, 2012, pp. 301-312.
4. S. Liu et al., ''Flikker: Saving Refresh-Power in Mobile Devices Through Critical Data Partitioning,'' *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2011, pp. 213-224.
5. L.N. Chakrapani et al., ''Ultra-Efficient (Embedded) SOC Architectures Based on Probabilistic CMOS (PCMOS) Technology,'' *Proc. Conf. Design, Automation and Test in Europe* (DATE 06), European Design and Automation Assn., 2006, pp. 1110-1115.
6. R. Hegde and N. R. Shanbhag, ''Energy-Efficient Signal Processing via Algorithmic Noise-Tolerance,'' *Proc. Int'l Symp. Low Power Electronics and Design* (ISLPED 99), ACM, 1999, pp. 30-35.
7. L. Leem et al., ''ERSA: Error Resilient System Architecture for Probabilistic Applications,'' *Proc. Conf. Design, Automation and Test in Europe* (DATE 10), European Design and Automation Assn., 2010, pp. 1560-1565.
8. S. Narayanan et al., ''Scalable Stochastic Processors,'' *Proc. Conf. Design, Automation and Test in Europe* (DATE 10), European Design and Automation Assn., 2010, pp. 335-338.
9. A. Sampson et al., ''EnerJ: Approximate Data Types for Safe and General Low-Power Computation,'' *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 11), 2011, pp. 164-174.
10. W. Baek and T.M. Chilimbi, ''Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 10), ACM, 2010, pp. 198-209.
11. A.R. Putnam et al., ''CHiMPS: A High-Level Compilation Flow for Hybrid CPU-FPGA Architectures,'' *Proc. ACM/SIGDA 16th Int'l Symp. Field Programmable Gate Arrays* (FPGA 08), ACM, 2008, doi:10.1145/1344671.1344720.
12. R. Razdan and M.D. Smith, ''A High-Performance Microarchitecture with Hardware-Programmable Functional Units,'' *Proc. 27th Ann. Int'l Symp. Microarchitecture,* ACM, 1994, pp. 172-180.
13. G. Venkatesh et al., ''Conservation Cores: Reducing the Energy of Mature Computations,'' *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2010, pp. 205-218.
14. V. Govindaraju et al., ''Dynamically Specialized Datapaths for Energy Efficient Computing,'' *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture,* IEEE CS, 2011, pp. 503-514.
15. S. Gupta et al., ''Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing,'' *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2011, pp. 12-23.
16. K. Przytula and V.P. Kumar, eds., *Parallel Digital Implementations of Neural Networks,* Prentice Hall, 1993.
17. B.E. Boser et al., ''An Analog Neural Network Processor with Programmable Topology,'' *IEEE J. Solid-State Circuits,* Dec. 1991, pp. 2017-2025.
18. A. Hashmi et al., ''A Case for Neuromorphic ISAs,'' *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2011, pp. 145-158.
19. T. Chen et al., ''BenchNN: On the Broad Potential Application Scope of Hardware Neural Network Accelerators,'' *Proc. IEEE Int'l Symp. Workload Characterization,* IEEE CS, 2012, pp. 36-45.
20. O. Temam, ''A Defect-Tolerant Accelerator for Emerging High-Performance Applications,'' *Proc. 39th Ann. Int'l Symp. Computer Architecture* (ISCA 12), IEEE CS, 2012, pp. 356-367.

Back-propagation is a gradient-descent algorithm that iteratively adjusts the neural network's weights according to each input-output pair.

Neural network topology selection. In addition to running back-propagation, 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 percent of the observed data) and a test set (the remaining 30 percent). 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

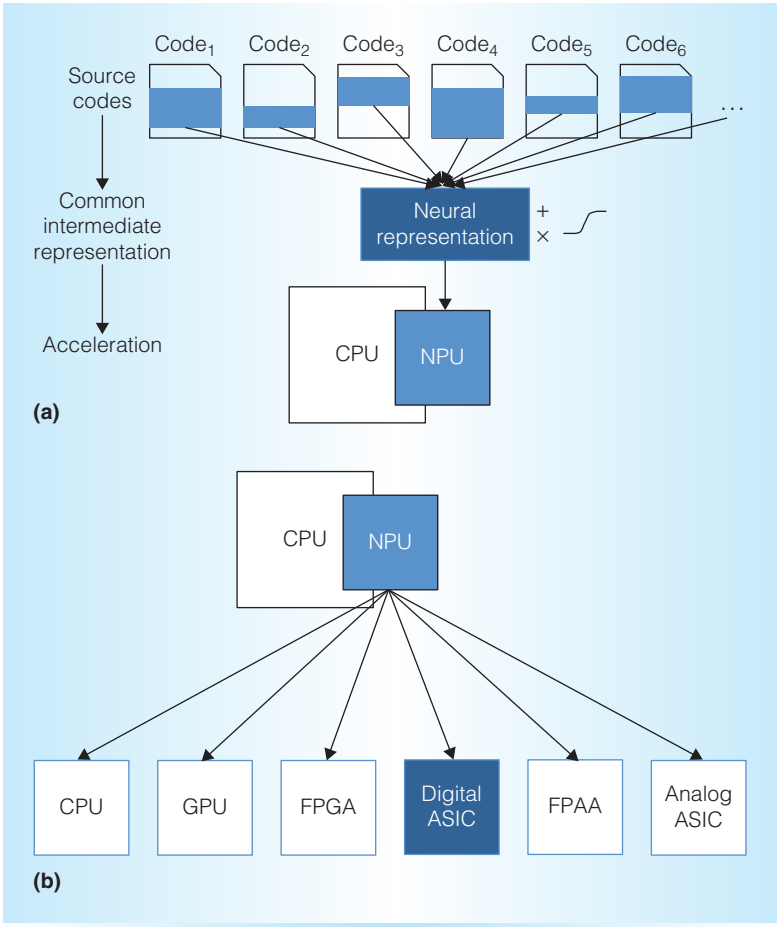

Figure 2. The Parrot algorithmic transformation converts different regions of code to a common neural intermediate representation that constitutes simple and highly parallel operations. Neural networks as a common representation (with structured, simple, and fine-grained parallel operations) enable acceleration of diverse applications using a single reconfigurable NPU. Because different applications require different neural topologies, the NPU is a reconfigurable accelerator (a). There are many possibilities for NPU implementation. This article focuses on a precise digital application-specific integrated circuit (ASIC) design (b).

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. Our 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 offsite training, multiple deployed application instances could centralize their training to increase input space coverage.

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.

Architecture design for NPU acceleration

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

ISA support for NPU acceleration

The NPU is a variable-delay, tightly coupled accelerator that communicates with the rest of the core via first-in, first-out (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 can be parameterized with an operand that identifies the target NPU.

- `enq.c %r` enqueues the value of the register *r* into the configuration FIFO.
- `deq.c %r` dequeues a configuration value from the configuration FIFO to 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—the 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.

Neural processing unit

As Figure 2b illustrates, there are many possibilities for the NPU implementation. Neural networks have previously been implemented in software on the CPU or GPU, on FPGAs, in digital ASICs, and even in analog circuitry or field-programmable analog arrays (FPAAs). We designed a digital NPU circuit that operates at the same voltage and frequency as the main core.[17] This implementation represents a reasonable tradeoff between efficiency and complexity. However, we believe that analog NPUs have significant potential, and we plan to explore them in future work.

Evaluation

To evaluate the Parrot transformation's effectiveness, 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 computation accuracy tradeoff. We perform a sensitivity analysis to examine the effect of the NPU processing-engine (PE) count and communication latency on the performance benefits. See our paper for MICRO 2012 for the details of our evaluation.[17]

Benchmarks

We used a suite of approximation-tolerant applications from diverse domains to evaluate our technique's broad applicability, including

- `fft` (Radix-2 Cooley-Tukey fast Fourier transform),
- `inversek2j` (inverse kinematics for two-joint arm),
- `jmeint` (triangle intersection detection),
- `jpeg` (image encoding),
- `kmeans` (clustering), and
- `sobel` (edge detection).

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.[3,6,7,9,10,15] To assess the effect of the Parrot transformation perceptually, we selected a number of benchmarks that generate image outputs. We didn't reject any of the applications on the basis of performance, energy, or accuracy shortfalls.

Code annotation. We annotated each benchmark's C source code as described earlier by identifying a single pure function with fixed-size inputs and outputs. We made no algorithmic changes to the benchmarks to accommodate the Parrot transformation. In some cases, we had more than one choice for the target code selection, and multiple NPUs might even have been beneficial for some programs. For the purposes of this evaluation, however, we selected a single target region per benchmark that was easy to identify, frequently executed 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.

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 free of side effects, and the number of inputs and outputs is statically identifiable.

Training data. To train the NPU for each application, we used either typical program inputs (for example, sample images) or 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 used an application-specific error metric to assess each benchmark's output quality. In all cases, we compared the original untransformed application's output to that of the transformed application. For `fft` and `inversek2j`, which generate numeric outputs, we measured the average relative error. The benchmark `jmeint` calculates whether two 3D 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.

Application average error rates range from 3 to 10 percent. This quality-of-service loss is commensurate with other work on quality tradeoffs.[3,9,16,18]

Experimental setup

We use cycle-accurate simulation and energy modeling to evaluate the performance and energy effects of the Parrot transformation and NPU acceleration.

Simulation. We use the MARSSx86 cycle-accurate x86-64 simulator to evaluate the performance effect of the Parrot transformation and NPU acceleration.[19] 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 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 program's cycle-accurate simulation. The resulting statistics are sent to a modified version of McPAT to estimate each execution's energy consumption. We model the energy consumption of an 8-PE NPU using the results from McPAT and CACTI 6.5 for memory arrays, buses, and steering logic. We use the results from Galal and Horowitz to estimate the energy of multiply-and-add operations.[20] 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 2,080 MHz frequency and $V_{DD} = 0.9$ V settings because the energy results in Galal and Horowitz use this frequency and voltage setting.

Experimental results

Figure 3a shows the application speedup when an 8-PE NPU replaces each benchmark's target function. The rest of the code runs on the core. The baseline executes the entire untransformed benchmark on the CPU. The plots also show the potential available speedup, which is the hypothetical speedup if the NPU takes zero cycles for computation. Among the benchmarks, `inversek2j` sees the highest speedup (11.1×) because the Parrot transformation substitutes the bulk of the application with a relatively small neural network (2 → 8 → 2). On the other hand, `kmeans` sees a 24 percent slowdown even though it shows a potential speedup of 20 percent 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 neural network ($6 \rightarrow 8 \rightarrow 4 \rightarrow 1$) 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 3b shows each benchmark's energy reduction. 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 a significant portion of dynamic instructions that otherwise would go through power-hungry stages of the out-of-order pipeline. The reduction in the number of dynamic instructions, together with the NPU's energy-efficient design, yields a 3.0× average application energy reduction.

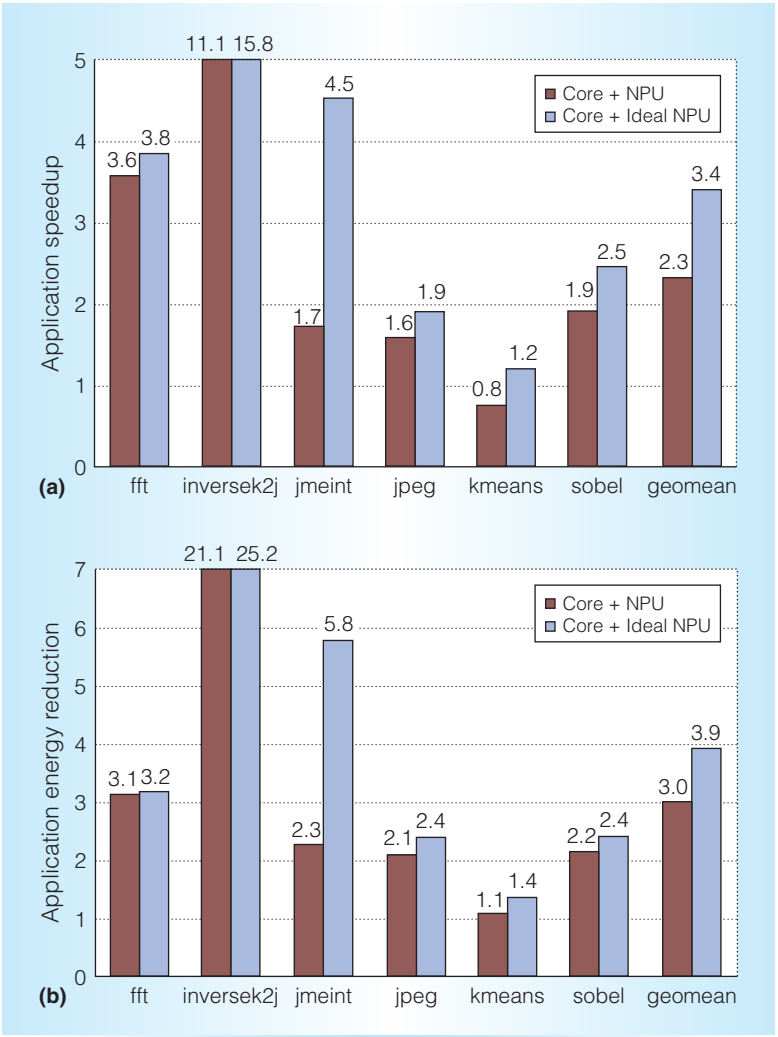

Figure 3. Performance and energy improvements. Total application speedup with an 8-processing-engine (8-PE) NPU (a). Total application energy saving with an 8-PE NPU (b).

Traditionally, hardware implementations of neural networks have been confined to specific classes of learning applications. In this article, we show that the potential exists to use them to accelerate general-purpose code that can tolerate small errors by introducing and defining the Parrot algorithmic transformation. As this work demonstrates, the Parrot algorithmic transformation, which structurally and semantically changes the code, can yield significant gains both in performance and efficiency. 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 digital and analog domains. MICRO

Acknowledgments

We thank the anonymous reviewers for their valuable comments. We also thank Brandon Lucia, Jacob Nelson, Ardavan Pedram, Mike Schlansker, 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. H. Esmaeilzadeh et al., "Dark Silicon and the End of Multicore Scaling," *Proc. 38th Ann. Int'l Symp. Computer Architecture* (ISCA 11), ACM, 2011, pp. 365-376.
2. R. Hameed et al., "Understanding Sources of Inefficiency in General-Purpose Chips," *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM, 2010, pp. 37-47.

3. A. Sampson et al., ''EnerJ: Approximate Data Types for Safe and General Low-Power Computation,'' *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 11), 2011, pp. 164-174.
4. Y. Fang, H. Li, and X. Li, ''A Fault Criticality Evaluation Framework of Digital Systems for Error Tolerant Video Applications,'' *Proc. Asian Test Symp.* (ATS 11), IEEE CS, 2011, pp. 329-334.
5. M. de Kruijf and K. Sankaralingam, ''Exploring the Synergy of Emerging Workloads and Silicon Reliability Trends,'' *Proc. IEEE 9th Workshop Silicon Errors in Logic—System Effects,* 2009.
6. X. Li and D. Yeung, ''Exploiting Soft Computing for Increased Fault Tolerance,'' *Workshop Architectural Support for Gigascale Integration,* 2006.
7. C. Alvarez et al., ''Fuzzy Memoization for Floating-Point Multimedia Applications,'' *IEEE Trans. Computers,* July 2005, pp. 922-927.
8. M. de Kruijf, S. Nomura, and K. Sankaralingam, ''Relax: An Architectural Framework for Software Recovery of Hardware Faults,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), 2010, pp. 497-508.
9. H. Esmaeilzadeh et al., ''Architecture Support for Disciplined Approximate Programming,'' *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2012, pp. 301-312.
10. L. Leem et al., ''ERSA: Error Resilient System Architecture for Probabilistic Applications,'' *Proc. Conf. Design, Automation and Test in Europe* (DATE 10), European Design and Automation Assn., 2010, pp. 1560-1565.
11. L.N. Chakrapani et al., ''Ultra-Efficient (Embedded) SOC Architectures Based on Probabilistic CMOS (PCMOS) Technology,'' *Proc. Conf. Design, Automation and Test in Europe* (DATE 06), European Design and Automation Assn., 2006, pp. 1110-1115.
12. S. Gupta et al., ''Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing,'' *Proc. 44th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM, 2011, pp. 12-23.
13. G. Venkatesh et al., ''Conservation Cores: Reducing the Energy of Mature Computations,'' *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, 2010, pp. 205-218.
14. V. Govindaraju et al., ''Dynamically Specialized Datapaths for Energy Efficient Computing,'' *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture,* IEEE CS, 2011, pp. 503-514.
15. S. Sidiroglou-Douskos et al., ''Managing Performance vs. Accuracy Trade-offs with Loop Perforation,'' *Proc. 13th European Conf. Foundations of Software Eng.,* ACM, 2011, pp. 124-134.
16. W. Baek and T.M. Chilimbi, ''Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 10), ACM, 2010, pp. 198-209.
17. H. Esmaeilzadeh et al., ''Neural Acceleration for General-Purpose Approximate Programs,'' *Proc. 45th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2012, pp. 449-460.
18. S. Misailovic et al., ''Quality of Service Profiling,'' *Proc. 32nd ACM/IEEE Int'l Conf. Software Engineering—Vol. 1* (ICSE 10), ACM, 2010, pp. 25-34.
19. A. Patel et al., ''MARSSx86: A Full System Simulator for x86 CPUs,'' *Proc. 48th ACM/EDAC/IEEE Design Automation Conf.,* IEEE CS, 2011, pp. 1050-1055.
20. S. Galal and M. Horowitz, ''Energy-Efficient Floating-Point Unit Design,'' *IEEE Trans. Computers,* July 2011, pp. 913-922.

Hadi Esmaeilzadeh is a PhD candidate in the Department of Computer Science and Engineering at the University of Washington. His research interests include developing new technologies and cross-stack solutions to improve the performance and energy efficiency of computer systems for emerging applications. Esmaeilzadeh has an MS in computer science from the University of Texas at Austin and an MS in electrical and computer engineering from the University of Tehran.

Adrian Sampson is a PhD student in the Department of Computer Science and Engineering at the University of Washington, where he works on architecture and programming languages. His research focuses

on disciplined approximate computing, a set of techniques that safely trade accuracy for efficiency. Sampson has an MS in computer science from the University of Washington.

Luis Ceze is an associate professor in the Computer Science and Engineering Department at the University of Washington. His research focuses on computer architecture, programming languages, and operating systems to improve multiprocessor systems' programmability, reliability, and energy efficiency. Ceze has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of IEEE, the ACM, and Usenix.

Doug Burger is the director of client and cloud applications at Microsoft Research, where he manages strategic research projects covering new user interfaces, datacenter specialization, cloud architectures, and platforms that support personalized online services. Burger has a PhD in computer science from the University of Wisconsin. He is a fellow of IEEE and the ACM.

Direct questions and comments about this article to Hadi Esmaeilzadeh, Computer Science and Engineering, Box 352350, Seattle, WA 98195; hadianeh@cs.washington.edu.