Towards Neural Acceleration for General-Purpose Approximate Computing

Hadi Esmaeilzadeh[†] Adrian Sampson[†] Luis Ceze[†] Doug Burger[‡]

[†]University of Washington [‡]Microsoft Research

{hadianeh,asampson,luisceze}@cs.washington.edu dburger@microsoft.com

Abstract

Energy efficiency is becoming crucial to realizing the benefits of technology scaling. We introduce a new class of low-power accelerators called Neural Processing Units (NPUs). Instead of being programmed, NPUs *learn* to behave like general-purpose code written in an imperative language. After a training phase, NPUs mimic the original code with acceptable accuracy. We describe an NPU-augmented architecture design incorporating a digital neural network implementation and a mechanism for invoking it from the main core. Simulation results show average speedups and energy savings both on the order of 2× with little quality loss for programs from diverse domains including signal processing, gaming, graphics, compression, machine learning, and image processing.

1. Introduction

The need for performance and energy efficiency in light of technology scaling limitations (*dark silicon* [8]) has motivated a drive towards specialization and acceleration. Recent work has also begun to exploit the fact that many applications can tolerate imprecise computation, which can be cheaper to provide. We leverage these two trends and introduce Neural Processing Units (NPUs), a new class of configurable accelerators for approximate computation.

Many classes of computations do not require universal precision. Image rendering, signal processing, augmented reality, and speech recognition can tolerate errors in substantial portions of their computation [6, 16, 24]. Architectures that provide these "soft" applications with optional low-precision operation can save significantly in both time and energy [6, 9, 18]. However, these previous approaches are limited by the costs of bookkeeping in general-purpose computing: instruction fetch, decode, and scheduling; memory accesses; loops and control flow; etc.

We introduce NPUs as a new class of accelerators without the overheads intrinsic to traditional general-purpose CPUs. NPUs act as accelerators for approximate code, implementing imprecise computations that mimic portions of the application where inaccuracy is tolerable. By completely replacing certain complex computations, NPUs can help reduce control flow costs that prior approaches to approximate computing or acceleration fail to address.

Unlike traditional accelerators and configurable computing units, NPUs are not programmed: instead, they *learn* to emulate the original computation using training input and output samples. The target code is a "black box" from the NPU's perspective: the algorithm and its implementation are irrelevant when training the accelerator. Hardware neural networks are fast and low-power, so executing a trained NPU can be vastly more efficient than running the original code. While previous work on efficient hardware neural networks has mainly considered applying them for learning tasks, NPUs are able to leverage them to accelerate general-purpose imperative code.

2. The NPU Program Transformation

Figure 1 depicts the *NPU transformation,* where a portion of a program's control flow graph is substituted with an invocation of a neural network. The transformation is transparent from the programmer's perspective: the system automatically trains the neural network and transforms the code to take advantage of it. The training process treats the code as a black box, so while the input and output of the target code must be in a form that can be learned by a neural network, the target code's *implementation* is unrestricted.

2.1 Applicability

Neural networks are intrinsically imprecise and will not perfectly imitate general imperative code. Therefore, programs that take advantage of them must incorporate application-level tolerance of imprecision. Prior work has explored this application-level tolerance in the context of faulty architectural components and approximate algorithms [6, 24, 25]. The NPU transformation applies to applications containing code whose output can be imprecise.

As an 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. Because the Sobel filter is fundamentally an approximation, slight errors in its computation are unlikely to cause major degradation of edge detection quality. In this example, an NPU can be used to replace the Sobel filter computation and accelerate the edge detection algorithm.

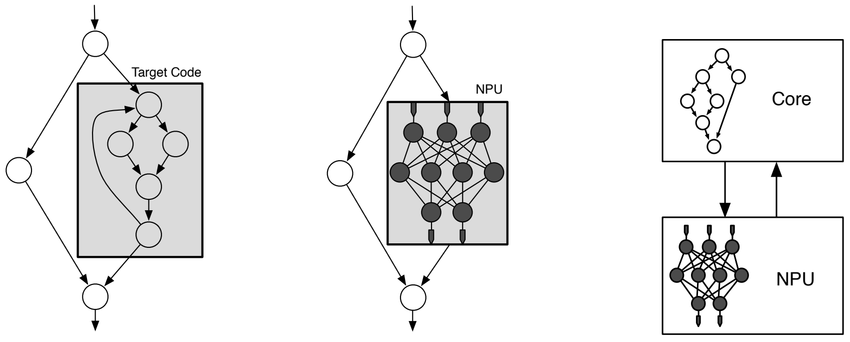
To be most effective, NPUs must replace an expensive function: one that would require many CPUs cycles to compute precisely. Ideally, the target function includes loops and other complex control flow—the cost of this complexity can be elided when computing on an NPU. The profitability of NPUs for smaller functions depends on the overhead of their implementation; we explore this trade-off in our evaluation (Section 5). The function-approximation properties of neural networks rely on their ability to interpolate, generalizing from examples to other inputs. Hence, NPUs are best applied to code that resembles a smooth function, for which new input/output pairs will resemble training data. NPUs are less likely to effectively approximate "chaotic" code, in which even large training sets can fail to represent the function's full range of behavior.

2.2 Programming Model

The use of NPUs has two implications for the programming model: the program must be able to tolerate imprecision in part of its computation; and it must be possible to isolate a portion of code for offloading to the NPU.

Approximation. Prior work has developed programming models that incorporate approximation in disciplined ways to allow software to remain robust in the face of errors in certain parts of the program [6, 24]. These languages allow the programmer to specify code or data that may be executed with relaxed semantics where imprecision is expressly permitted. The feasibility of NPUs depends on such a programming model: the NPU transformation can only apply to code when the programmer has allowed to be imprecise. Here, we use EnerJ [24] as our programming language.

Target code identification. To apply the NPU transformation, we need to identify a code slice that: (1) is *pure* and has no side effects; (3) is expensive enough; and (2) has fixed-size inputs and outputs. the code may not dynamically write an unbounded amount of data to a variable-length array.



(a) Original CFG (b) Transformed program (c) Architecture with NPU

Figure 1: NPU transformation takes an input program (a) and replaces a portion of its CFG (the *target code*) with an invocation of a neural network (b). During execution (c), the program runs mainly on the CPU, sending inputs to and receiving outputs from the NPU.

```
int sobelIntensity(int p[3][3]) {
    int x, y, r;
    x = (p[0][0] + 2 * p[0][1] + p[0][2]);
    x -= (p[2][0] + 2 * p[2][1] + p[2][2]);
    y = (p[0][2] + 2 * p[1][2] + p[2][2]);
    y -= (p[0][0] + 2 * p[1][1] + p[2][0]);
    x = pow(x, 2);
    y = pow(y, 2);
    r = (int)pow(x + y, 0.5);
    if (r >= 256)
        r = 255;
    return r;
}
```

(a) Imperative implementation of the Sobel filter

```
int sobelIntensity(int p[3][3]) {
    int r;
    enq.d p[0][0];
    enq.d p[0][1];
    enq.d p[0][2];
    enq.d p[2][0];
    enq.d p[2][1];
    enq.d p[2][2];
    enq.d p[1][1];
    enq.d p[2][0];
    deq.d r;
    return r;
}
```

(b) Replacement NPU invocation

Figure 2: Example imperative code and its replacement NPU invocation. In (a), the computation is implemented in a C-like language. In (b), we show the assembly instructions that invoke the NPU.

These constraints are not specific to the NPU transformation and would be appropriate for any code-centric approach to approximate computing, such as Relax [6]. Figure 2a shows an example computation that satisfies the constraints. This function implements a Sobel filter as part of an edge detection algorithm. The application can tolerate imprecision in the result value, r. The argument and result types have fixed size: nine words and one word respectively. Finally, the body of the computation is costly enough that an NPU invocation may be faster. More complex target code including heavier control flow would represent even greater potential benefit.

Similar candidate code may be discovered automatically via a static analysis over code written in an approximation-aware language. In this paper, however, we consider an approach based on programmer-provided hints: the programmer explicitly annotates a function in a program as a candidate for NPU replacement. In our experience with approximate programs, computations satisfying these constraints are easy to identify by hand (see Section 5). However, while this approach limits the NPU transformation to discrete code regions, a more sophisticated analysis would allow finer-grained *slices* of code—not just explicit functions—to be replaced.

2.3 Training

Before the program can use an NPU at runtime, the NPU must be trained on executions of the target code to mimic the code. Training may occur *off-line*, during compilation or deployment, or *on-line*, in parallel with the *in-vivo* execution of the target code. While on-line training may be appropriate in cases where appropriate test inputs are unavailable, this paper focuses on off-line training.

2.4 Execution

At run time, an NPU-aware program must configure the NPU according to the parameters derived during the training phase. An ISA extension (Section 3.1) allows the program to communicate the neural network's topology and weights to the NPU before it first invokes the NPU (e.g., when the program is started). Then, when the target function would otherwise have been invoked, the program sends inputs to the NPU and collects the resulting outputs as shown in Figure 2b. The compiler can omit the target code and the variables used for its intermediate values.

3. Architecture

There are a variety of different implementation strategies for NPUs with varying trade-offs in performance, power, area, and complexity. Both hardware and software neural network implementations are possible. In software, traditional CPU and highly parallel GPU implementations may be useful if the replaced imperative code is very coarse-grained. In systems equipped with configurable computing fabrics, NPUs may be implemented on FPGAs [27] or FPAAs [21]. Neural network ASIC designs are likely to be even more power-efficient and low-latency; prior work has explored both digital [7] and analog [3] neural network implementations. In this section, we describe an ASIC design for an NPU operating at the same critical voltage as the main core.

3.1 NPU ISA Support

The NPU is a variable-delay functional unit that communicates with the rest of the core via FIFO queues. The CPU–NPU interface consists of three queues: one for sending and retrieving the configuration, one for sending the inputs, and one for retrieving the neural network's outputs. The ISA is extended with four NPU queuing instructions. 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 immediate operand reflecting the NPU configuration's unique identifier.

- **enq.c %reg**: enqueue register into the NPU config FIFO.
- **deq.c %reg**: dequeue a value from the NPU config FIFO.
- **enq.d %reg**: enqueue register into the NPU input FIFO.
- **deq.d %reg**: dequeue from the NPU output FIFO to register.

To communicate with the NPU, the compiler emits a series of queueing instructions. To set up the NPU at program start time, the program executes a series of enq.c instructions to send configuration parameters—number of inputs and outputs, network topology, and neuron 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 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 output values from the invocation.

3.2 Integrating the NPU Into the Processor Pipeline

We discuss the microarchitectural support required for integrating the NPU with an speculative OoO pipeline.

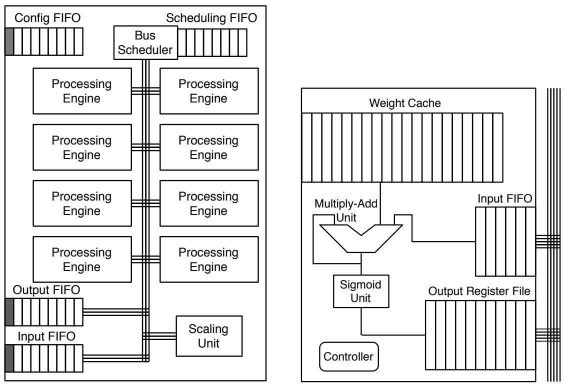
Instruction scheduling and issue. To guarantee correct communication with the NPU, the processor must issue NPU instructions in order. The renaming login and the scheduler will therefore treat all NPU instructions as dependent. Furthermore, the scheduler only issues an enqueue instruction if the corresponding FIFO is not full. Similarly, the dequeue instruction is only issued if the corresponding FIFO is not empty. To ensure correct speculative execution, the output FIFO maintains two head pointers: a speculative head and a non-speculative head. The speculative head pointer is updated when a dequeue instruction is issued; the non-speculative head pointer is only updated when the instruction commits.

Speculative execution and pipeline flushes. The processor can issue the enq.d and deq.d instructions speculatively. In case of a branch or dependence misspeculation, it must be possible to roll back the speculative operations. In the event of a pipeline flush, 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 tail pointer of the output FIFO to invalidate any outputs that are based on the invalidated inputs.

Commit. Since the enqueue and dequeue instructions can be executed speculatively, the head pointer of the input FIFO can only be updated—and consequently the entries recycled—if: (1) the enqueue instruction commits; and (2) the NPU completes 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. The entry will only be recycled when the subsequent deq.d instruction commits.

3.3 Reconfigurable NPU Architecture

This section details the design of the NPU itself as shown in Figure 3a. The design is reconfigurable: it can implement many different neural network topologies. As shown in Figure 3a, the NPU contains eight identical processing engines (PEs) and one scaling unit. Although the design can scale to larger and smaller numbers of PEs, we find that eight PEs can exploit the parallelism in the neural networks while larger NPUs exhibit diminishing returns in performance. The scaling unit scales the neural network's inputs and outputs using scaling factors defined in the NPU configuration process. Figure 3b shows the internal structure of a single PE.



(a) An 8-PE NPU (b) A single processing engine

Figure 3: Schematics for an NPU with 8 processing engines (PEs) and for a single PE.

The NPU is statically scheduled. The scheduling information is part of the NPU configuration, which is based on the neural network topology derived during the training process. Recall that the program executes enq.c instructions that send this configuration to the NPU. In the NPU's schedule, each neuron in the neural network is assigned to one of the eight PEs. The 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 scheduling FIFO (shown in Figure 3a). Each entry in this FIFO schedules the bus to broadcast a value from a PE or the input FIFO to a set of destination PEs or the output FIFO. A scheduling FIFO 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 3b). The destination is either the output FIFO or a set of PEs.

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 $y = \mathrm{sigmoid}(\sum_i (x_i \times w_i))$ where x_i is an input to the neuron and w_i is its corresponding weight. As shown in Figure 3b, the weight cache (a circular buffer) stores the weights.

4. Compilation and Training

To take advantage of an NPU, a program must train the neural network to emulate the target code. First, the compiler observes the inputs and outputs of the target code. Then, using this collected data, it executes a neural network's supervised learning algorithm. Finally, the compiler produces a new binary that invokes the NPU instead of the original code. Except for annotating the target code, this entire compiltion process is automatic.

4.1 Observation Phase

In the first phase, the compiler collects input/output pairs for the target code that reflect real program executions. This in-context observation allows us to train the NPU on a realistic data set, improving its recall accuracy during execution. The compiler produces an instrumented binary for the source program that includes probes on the input and output of the target code. Specifically, the candidate function records its arguments and return value at every execution. The output of this phase is a training data set.

4.2 Training Phase

Next, the compiler uses the training examples to produce a neural network configuration to be used as an NPU. 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. To train an MLP, the compiler must select parameters—a learning rate, training epoch count, an expected error value—that balance between overtraining and accuracy. Empirically, we find that a learning rate of 0.3, an epoch count of 5000, and an expected error of 0.01 work well for a variety of applications. In addition to running the backpropagation training algorithm [23], this phase must select an optimal network topology that balances between accuracy and efficiency. An MLP consists of a 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.

We use a simple heuristic to choose the topology, guided by the mean squared error (MSE) of the neural network when recalling a 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* and an *evaluation set*. The training set, which consists of 90% of the observed data, is used to drive the supervised learning algorithm. Once the neural network is trained, its accuracy is assessed over the evaluation set. To choose an acceptable network topology, the compiler starts with two neurons and a single hidden layer. It trains this network using the above parameters and evaluates its MSE when recalling the evaluation data. If the neural network exceeds the target error rate, it doubles the number of neurons in the hidden layer and trains again. If the number of neurons in the last hidden layer is greater than the number of inputs to the layer, the compiler adds a new hidden layer. This process repeats until the desired error level is met or the neural network reaches the maximum size allowed by the NPU.

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.

4.3 Scheduling the Neural Network on the NPU

Given the results of the training phase, the compiler configures the NPU to execute the discovered neural network topology. As Section 3.3 describes, the compiler must translate the neural network parameters into a low-level NPU configuration including a static schedule for the NPU's operation.

4.4 Embedding Parameters in the Binary

Finally, the compiler produces a new binary that invokes the NPU. The target code is not included in this binary; instead, the compiler emits special instructions (Section 3.1) to configure and execute the neural network. Before a program uses the NPU for the first time, it executes the enq.c instructions produced by the scheduling algorithm to set up the neural network according to the topology and neuron weights found during training. To invoke the replaced function, the compiler emits end.q and deq.d instructions at every call site as illustrated in Figure 2b.

5. Evaluation

Table 1 lists the benchmarks used in this evaluation. The application domains—signal processing, gaming, compression, machine learning, 3D graphics, 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, 9, 17, 18, 24, 25]. To select our benchmarks, we began with the applications used in the evaluation of EnerJ [24] and Truffle [9]. Of these, **fft**, **jmeint**, and **raytracer** contained code segments that were amenable to the NPU transformation. We identified three new approximate applications (**jpeg**, **kmeans**, and **sobel**) and found that each of them had NPU-amenable code segments. We did not reject any applications based

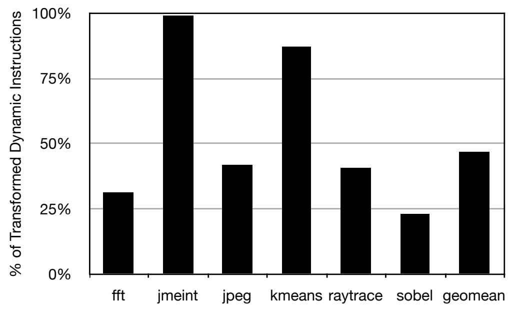


Figure 4: Percentage of dynamic instructions subsumed by NPU.

Table 2: NPU microarchitectural parameters.

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

on performance, energy, or accuracy shortfalls. Like previous work, we evaluate potential energy and performance gains in the context of the resulting loss of precision (or reduction in quality of service).

The Java source code for each benchmark was annotated as described in Section 2.2: we marked a single pure function with fixed-size inputs and outputs for offloading to the NPU. No substantial algorithmic changes were made to the benchmarks to allow them to accommodate the NPU transformation. Qualitatively we found it straightforward to identify a reasonable candidate function for the NPU transformation in each benchmark.

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. In **jpeg**, the NPU transformation subsumes the discrete cosine transform and quantization phases, each of which contains multiple function calls and loops. In **kmeans**, the target code includes centroid and distance calculations, which themselves include complex control structures. Because the target code in this benchmark is inside a loop, part of the output of one NPU invocation is used as the input to the next invocation. In **raytracer** and **sobel**, the target code is simpler and finer-grained, consisting mainly of arithmetic and logical operations and some if/then conditionals. In each case, the target code is side-effect-free and the number of live-ins and live-outs are statically identifiable. The NPU's scaling unit normalizes every input and output value.

Subsumed imperative code. Figure 4 depicts the proportion of each benchmark's dynamic execution that is replaced with NPU invocations in our experiments. The potential benefit of NPU-based execution is directly related to the amount of CPU work that is elided. Specifically, **jmeint** exhibits the greatest potential for benefit: nearly all of its dynamic instructions are in the target region for the NPU transformation. This is because **jmeint** is in a sense an ideal case: the entire algorithm has a fixed-size input (the vertex coordinates for two 3D triangles), fixed-size output (a single boolean indicating whether the triangles intersect), and a tolerance for imprecision. In contrast, **sobel** is representative of applications where NPUs apply more locally: the target code is "hot" but only consists of a few arithmetic operations and relatively simple con-

Table 1: Benchmarks used in the evaluation. The topology of the traind neural networks and their the mean squared error

Benchmark	Description	Type	Input Set	NN Topology	MSE
fft	SciMark2 benchmark: fast Fourier transform	Signal Processing	Vector of random floating-point numbers	$5 \to 8 \to 1$	0.0041
jmeint	jMonkeyEngine game framework: triangle intersection detection	3D Gaming	100 pairs of 3D triangle coordinates	$18 \to 32 \to 8 \to 2$	0.0053
jpeg	JPEG encoding	Compression	220×220-pixel color image	$64 \to 16 \to 64$	0.0089
kmeans	K-means clustering	Machine Learning	100 random two-coordinate data points	$5 \to 4 \to 2$	0.0001
raytracer	3D image renderer	3D Graphics	3D scene rendered to 450-pixel output image	$8 \to 8 \to 3$	0.0039
sobel	Sobel edge detector	Image Processing	220×220-pixel color image	$9 \to 16 \to 1$	0.0019

trol flow. Based on the results, both categories of application can benefit from NPUs, although **jmeint**'s improvements are greater.

Neural network topologies and error levels. Table 1 shows the neural network topology for each benchmark as discovered by the training process. In all cases, the neural network is a feed-forward multilayer perceptron. For example, for **jmeint**, the topology is $18 \to 32 \to 8 \to 2$, meaning that the neural network takes in 18 inputs, produces 2 outputs, and has two hidden layers with 32 and 8 neurons. The table shows the mean squared error (MSE) for each benchmark's neural network. The evaluation data is distinct from the training data. The NPU's output error affects the overall application quality of service differently for each benchmark. For example, with these error rates, **jmeint** exhibits a 7% misclassification rate and our two image analysis benchmarks, **sobel** and **jpeg**, produce images with less than 5% overall pixel value difference with respect to the precise execution. These error rates are in line with previous approximate computing evaluations [6, 9, 18, 24].

5.1 Experimental Setup

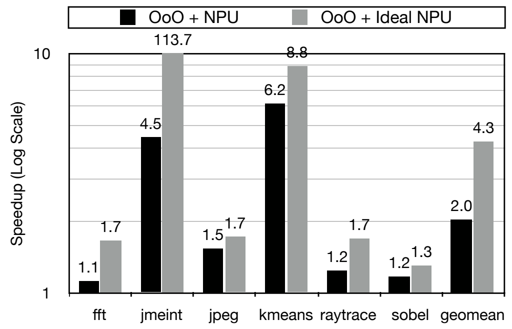
We simulate the execution of each benchmark to estimate runtime and energy consumption on platforms with and without NPUs. operations. We consider NPUs in conjunction with a 4-wide issue OoO core similar to Alpha 21264 [13] with L2 hit latency of 10 cycles and 200 cycles of memory access latency. We use a source-to-source transformation that instruments the benchmarks' Java code to emit an event trace including memory accesses, branches, and arithmetic operators. This source-level instrumentation is unaffected by the JIT, garbage collection, or other VM-level systems. Using a trace-based simulator, we generate architectural event statistics. The architectural simulator includes a cache simulation. The simulation process outputs detailed statistics, including the cycle count, cache hit and miss counts, and the number of functional unit invocations. We simulate each benchmark twice: once with the NPU-replaced code included and once with it excluded (i.e., to simulate offloading the functionality to the NPU). The trace-based CPU simulator is augmented with a cycle-accurate NPU simulator that also generates the statistics required for the NPU energy estimation.

The resulting statistics are sent to a modified version of McPAT [15] to estimate the energy consumption of each execution. We model the energy consumption of an 8-PE NPU using the results from CACTI 6.5 [19] and McPAT [15] for memory arrays, buses, and steering logic. We use the results from Galal et al. [11] to estimate the energy of multiply-and-add 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, 2080 MHz and 0.9 V.

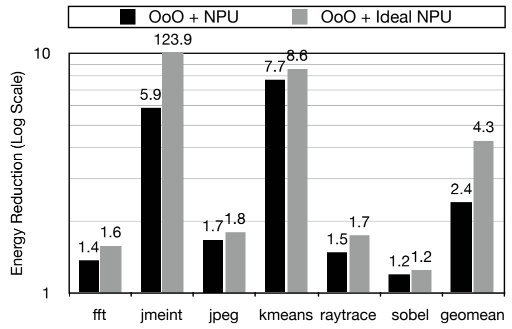
The modeled performance and energy for the NPU depends on the neural network topology. Complex networks, like the one for **jmeint**, consume more time and energy than smaller ones. Specifically, NPU execution times range from 14 cycles (for **kmeans**) to 330 cycles (for **jmeint**).

5.2 Speedup and Energy Savings

Figure 5a shows the speedup when an 8-PE NPU is used to replace the target code. The rest of the code runs on the general-purpose



(a) Speedup on OoO core



(b) Energy savings on OoO core

Figure 5: Performance and energy improvements. The "Ideal NPU" values indicate the hypothetical gains if the NPU were to take zero time and zero energy.

core. The baseline consists of 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 to perform a recall. On average, the benchmarks see a 2.0× speedup. Figure 5b shows the energy reduction for NPU-augmented design. Again, the baseline is the energy consumed by running the entire benchmark on the unmodified CPU and the "ideal" energy savings correspond to using a hypothetical zero-energy NPU. On average, the programs see a 2.4× energy reduction. These results indicate a significant speedup and energy reduction for an 8-PE floating-point digital NPU. The limit results, where we consider a hypothetical zero-energy instantaneous NPU, suggest that using an even more efficient NPU implementation can lead to even greater savings—up to about 4× performance and energy improvements on average.

6. Related Work

This work has three well-established categories of related work: *approximate computing*; *synthesis and configurable computing*; and *hardware neural networks*.

Many categories of "soft" applications have been shown to be tolerant to imprecision during execution. Prior work has explored relaxed hardware semantics and their impact on these applications, both as extensions to traditional architectures [6, 9, 18] and in the form of fully approximate processing units [4, 14, 20]. In contrast to the former, in which fine-grained portions of a mostly-precise execution are made approximate, NPUs can apply to coarse blocks of code or even entire algorithms. NPU invocations can subsume imperative control operations and other overheads of precise computation. The NPU design presented in this paper eliminates dynamic instruction scheduling in the replaced application code, leading to faster and more energy-efficient execution. In contrast to the latter category, in which entire processing units carry relaxed semantics and thus vastly different programming models, NPUs can be used transparently from traditional programs. No special code must be written to take advantage of the general-purpose approximate units; instead, using a learning mechanism, NPUs can accelerate existing applications with minimal programmer intervention. Some work has also exposed relaxed semantics in the programming language to give programmers control over the precision of software [2, 6, 24]. These proposals allow programmers to use approximation—such as NPUs—effectively.

NPUs extend prior work in the areas of configurable computing, synthesis, specialization, and acceleration that focuses on compiling traditional, imperative code into efficient hardware structures. Significant progress has been made on synthesizing efficient circuits and configuring FPGAs to accelerate general-purpose code [5, 10, 22]. Static specialization has also shown significant efficiency gains for irregular and legacy code [26]. 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.

Finally, our neural-network NPU design builds on research surrounding the implementation and exploitation of hardware neural network implementations, both analog [3] and digital [7]. Prior work has also proposed abstractions for exposing learning mechanisms to software [12]. Training-based acceleration represents a new way to use neural networks in general computations.

7. Conclusions

This paper showed how artificial neural networks can be used as fine-grain accelerators for a wide range of approximable codes. We introduced a specific design called Neural Processing Units and a compiler workflow that uses a programmer annotation to identify the function to accelerate. With that function, the compiler can automatically transform the code, train the NPU, and invoke the NPU at runtime, showing consistent performance improvements and energy savings both on the order of $2\times$. We showed that a simple algorithm can search the space of network topologies and produce neural networks with comparable error rates to previous, more conventional approaches to approximate computation. Traditionally, hardware implementations of neural networks have been confined to specific classes of learning applications. In this paper, we have shown that the potential exists to use them to accelerate significant chunks of general-purpose code that can tolerate small errors. This capability aligns with both transistor and application trends. Neural networks thus have the potential to form a new, widely used class of accelerators, similar to the roles that FPGAs and GPUs play today.

References

[1] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(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. Sckinger, J. Bromley, Y. Lecun, L. D. Jackel, and S. Member. An analog neural network processor with programmable topology. *J. Solid-State Circuits*, 26:2017–2025, 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] 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.

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

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

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

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

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

[11] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *IEEE Trans. Comput.*, 60(7):913–922, 2011.

[12] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti. A case for neuromorphic ISAs. In *ASPLOS*, 2011.

[13] R. E. Kessler. The Alpha 21264 Microprocessor. *MICRO*, 1999.

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

[15] 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.

[16] X. Li and D. Yeung. Exploiting soft computing for increased fault tolerance. In *ASGI*, 2006.

[17] X. Li and D. Yeung. Exploiting application-level correctness for low-cost fault tolerance. *J. Instruction-Level Parallelism*, 2008.

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

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

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

[21] S.-Y. Peng, P. Hasler, and D. Anderson. An analog programmable multidimensional radial basis function based classifier. *TCAS-I*, 54 (10):2148–2158, 2007.

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

[23] 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*, volume 1, pages 318–362. MIT Press, 1986.

[24] 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.

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

[26] 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.

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