

# Approximate Computing on Programmable SoCs via Neural Acceleration

Thierry Moreau    Jacob Nelson    Adrian Sampson  
Hadi Esmaeilzadeh\*    Luis Ceze

University of Washington

\*Georgia Institute of Technology

## Abstract

*Processor designs for portable, ubiquitous computing devices such as cell phones have widely incorporated hardware accelerators to support energy-efficient execution of common tasks. Porting applications to take advantage of these resources is often a difficult task due to the restricted programming model of the accelerator: FPGA-based acceleration, for instance, often requires the expertise of a hardware designer. Fortunately, many applications that can take advantage of accelerators are amenable to approximate execution, which prior work has shown can be exploited with a simple programming model.*

*This paper presents a comprehensive and mostly automatic framework that allows general-purpose approximate code to use programmable logic without directly involving hardware design. We propose SNNAP, a flexible FPGA-based neural accelerator for approximate programs. We identify the challenges associated with this approach and design a hardware framework that offers this capability. We measure a real FPGA implementation on a state-of-the-art programmable system-on-a-chip (PSoC) and show an average  $1.77\times$  speedup and  $1.71\times$  energy savings.*

## 1. Introduction

In light of diminishing returns from technology improvements on performance and energy efficiency [18, 25], researchers are actively seeking solutions that will provide possible paths forward. There are at least two clear trends emerging. One is the use of specialized logic in the form of accelerators [23, 24, 47, 48] or programmable logic [11, 36, 38]; and another is taking advantage of the broad array of applications that can tolerate quality degradations in return for performance and energy efficiency—a.k.a., *approximate computing*. Specialization leads to better efficiency by trading off flexibility for leaner logic and hardware resources, and approximate computing improves efficiency by embracing inaccurate behavior. The confluence of the two trends can lead to still more opportunities to improve efficiency.

One promising approach to specialization for energy efficiency is the incorporation of programmable logic. On-chip field-programmable gate arrays (FPGAs) have the potential to unlock order-of-magnitude energy efficiency gains when they are configured to offload work from the CPU [44]. Commercial system-on-a-chip parts that incorporate a large amount of programmable logic [2, 50] are beginning to appear. But

enabling mainstream programmers, who are accustomed to software and not hardware design, to write efficient FPGA configurations remains a challenge. High-level synthesis and C-to-gates compilers have been shown to work for some domains but have limited applicability [1, 16].

This paper explores an opportunity to use programmable logic to accelerate *approximate* programs without the need for per-application FPGA designs. Instead, we instantiate a flexible, high-performance neural network design in programmable logic. Recent work has shown how to use a neural network as a general accelerator for approximate programs [5, 10, 20, 46]. Our accelerator hardware, called SNNAP (systolic neural network accelerator in programmable logic), enables neural acceleration for heterogeneous systems augmented with FPGAs. SNNAP can be configured to accelerate a wide range of programs just by loading new weights into the accelerator—without reconfiguring the underlying FPGA fabric. This allows SNNAP to support new applications without requiring hardware design or synthesis.

Approximate programs can use SNNAP for acceleration via a compiler workflow that automatically configures the neural network’s topology and weights instead of the programmable logic itself. The key idea is to train a logical neural network to behave like regions of approximate code. Once the neural network is trained, the system no longer executes the original code and instead invokes the neural network model on a *neural processing unit (NPU)*. This leads to better efficiency because neural networks are amenable to efficient implementation in hardware [17, 29, 37, 42]. Prior work on NPUs [20] for general-purpose approximate computing, however, have assumed NPUs as fully custom logic tightly integrated with the processor core (evaluated in simulation). While this increases potential applicability, it limits adoption and requires deeper changes to existing core designs.

SNNAP’s adaptable neural network design offers several advantages compared to designing special custom logic for each region of code to be accelerated. First, the programming model and neural network training framework we employ frees the programmer from having to design logic. Second, a very diverse body of code can be accelerated with the same circuit design, avoiding expensive FPGAs reconfigurations. The only thing that changes between code regions are the neural network configuration parameters. Our NPU design offers fast reconfigurability without resynthesizing the circuit. Finally, this approach sidesteps the limitations of high-level

synthesis (C-to-gates) approaches. While there is likely some efficiency loss compared to synthesizing a specific design for each code region, our results show that it is viable and beneficial to do approximate neural-based acceleration on programmable logic.

The main contribution of this work is a comprehensive framework that allows general-purpose approximate code to exploit programmable logic without directly involving hardware design. We used a real FPGA for our implementation and actual measurements on a state-of-the-art programmable system-on-a-chip (PSoC). We identify two core challenges: data communication latency between the core and the programmable logic unit; and the frequency discrepancy between the programmable logic and the core. We address those challenges with careful design of a throughput-oriented interface and an architecture based on scalable systolic arrays.

To evaluate our framework, we ran a suite of approximate benchmarks on our design implemented in the SoPC. We found a average speedup of  $1.77\times$ , with the fastest benchmark achieving a  $17.95\times$  speedup and the slowest  $0.57\times$ . We obtained similar energy savings; the average was  $1.71\times$ , the maximum was  $17.44\times$ , and the minimum was  $0.56\times$ .

## 2. Programming

SNNAP accelerates regions of approximate code. This section describes its interfaces and programming model. The layers range from the instruction-level interface emitted by the compiler to an automatic transformation, termed *neural acceleration*, that invokes SNNAP transparently based on minimal program annotations.

### 2.1. Neural Acceleration Overview

While SNNAP can speed up explicit neural network invocations, it is most broadly applicable when it can automatically replace expensive, approximate code written in the source language. To enable software to exploit SNNAP automatically, we leverage the *neural acceleration* paradigm originally proposed by Esmaeilzadeh et al. [20]. Here we briefly recap the neural acceleration workflow.

The process begins with an approximation-aware programming language that marks some code and data as approximable. Language options include Relax’s code regions [15], EnerJ’s type qualifiers [40], Rely’s variable and operator annotations [7], or simple function annotations. In any case, the programmer’s job is to express where incorrect results are allowed and where they could break fundamental program invariants. The neural-acceleration compiler enumerates the implied regions of approximate code, termed *target regions*, and transforms each of them to use an abstract neural network. During the transformation process, the compiler uses a set of test inputs to repeatedly execute the program and collect input and output values for each target region. Standard training algorithms produce a neural network according to each input–

output data set and, finally, the compiler replaces the original code with an invocation of the learned neural network.

As an example, consider a program that filters each pixel in an image. The original code might resemble:

```
APPROX_FUNC double filter(double pixel);
...
for (int x = 0; x < width; ++x)
  for (int y = 0; y < width; ++y)
    out_image[x][y] = filter(in_image[x][y]);
```

where the `filter()` function is marked as approximable. A final transformed version of the code replaces the `filter` call with a call to invoke SNNAP (`nn_invoke`) and adds a call early in the program to set up the neural network for invocation (`nn_configure`):

```
nn_configure(...);
for (int x = 0; x < width; ++x)
  for (int y = 0; y < width; ++y)
    out_image[x][y] = nn_invoke(in_image[x][y]);
```

While the above example uses explicit function annotation, the technique extends to other generic approximate programming models where approximable regions are implicit. For example, in EnerJ [40], annotations express what may be approximated but not specifically how—the same annotation set can permit low-power functional units [19, 30, 49], unreliable storage [9, 19, 31, 41], and even algorithmic changes [43]. Neural acceleration fits into the array of approximation strategies when any region of code (a) has exclusively approximate effects, (b) has a fixed number of numeric live-ins and live-outs identifiable by the compiler, and (c) is computationally expensive according to a performance profile.

### 2.2. Low-Level Interfaces

While automatic transformation represents the highest-level interface to SNNAP, it is built on lower-level interfaces intended to be emitted by compilers or used directly by experts. This section details the instruction-level interface to SNNAP and a low-level library layered on top of it that makes its asynchrony explicit.

Unlike a low-latency circuit that can be tightly integrated with the pipeline, FPGA-based accelerators cannot afford to block program execution to compute each individual input. Instead, we architect SNNAP to operate efficiently on batches of inputs. Software groups together invocations of the neural network and ships them all simultaneously to the FPGA for pipelined processing. In this sense, SNNAP behaves as a *throughput-oriented* accelerator: it behaves most effectively when the program keeps it busy with a large number of invocations rather than when each individual invocation must complete quickly.

**Instruction-Level Interface** At the lowest level, the program invokes SNNAP by enqueueing batches of inputs, invoking the accelerator, and receiving a notification when the batch is complete. Specifically, the program writes all the inputs into a buffer in memory and uses the ARM `SEV` (send event) instruction to notify SNNAP. The accelerator then reads

the inputs from the CPU’s cache via the coherence interface and processes them, placing the output into another buffer. Meanwhile, the program issues an ARM `WFE` (wait for event) instruction to sleep until the neural-network processing is done and then reads the outputs.

**Asynchronous Call** The most flexible interface to SNNAP reflects its asynchronous nature by explicitly separating the invocation from the collection of results. The programmer calls `nn_send(x)` to build up a batch of inputs for SNNAP and later calls `nn_receive()` to request the computed outputs. These functions are responsible for implicitly building up batches, sending them to SNNAP, and buffering results.

Recall the above pixel-filtering example. The `nn_invoke` call inside the loop is equivalent to an `nn_send` immediately followed by an `nn_receive`, which permits no batching and no asynchronous execution. Instead, a more efficient solution uses two loops, one to send the inputs and one to receive the results:

```
for (int x = 0; x < width; ++x)
  for (int y = 0; y < width; ++y)
    nn_send(in_image[x][y]);
for (int x = 0; x < width; ++x)
  for (int y = 0; y < width; ++y)
    out_image[x][y] = nn_receive();
```

This asynchronous style lets the SNNAP runtime library build batches of pixels. In more sophisticated programs, it also lets SNNAP run in parallel with other program code that occurs between `nn_send` and `nn_receive` calls.

This style resembles *promise* or *future* constructs from parallel programming. An `nn_send` call enqueues an input value and, if the buffer then becomes full, sends the input batch and clears it. Each `nn_receive` call consumes and returns the output for the oldest unconsumed SNNAP invocation. (In other words, the *n*th receive call that the program executes corresponds to its *n*th send call.) If no outputs are available, the call blocks until a new result batch is received. If `nn_receive` is called before the input buffer is full, a partial batch is sent to SNNAP for computation to avoid deadlock.

This low-level, asynchronous interface is suitable as a compilation target and for expert programmers exploiting fine-grained task parallelism. The programmer does not explicitly write `nn_send` and `nn_receive` calls when using automatic program transformation; these calls are inserted by the compiler to replace approximate code blocks.

**Software Pipelining Optimization** In most cases, including the example above, the SNNAP invocation appears in a DOALL loop where each iteration is independent. In these cases, we can use standard loop unrolling and software pipelining optimizations to automatically make efficient use of SNNAP without programmer intervention. The transformation rewrites a DOALL loop to operate in batches using a sequence of send and receive calls. If `BATCH_SIZE` denotes the number of inputs per SNNAP invocation, then the original loop above is transformed to:

```
for (int x = 0; x < width; ++x)
  for (int y = 0; y < width; y += BATCH_SIZE) {
    nn_send(in_image[x][y]);
    nn_send(in_image[x][y + 1]);
    ...
    out_image[x][y] = nn_receive();
    out_image[x][y + 1] = nn_receive();
    ...
  }
```

In practice, many of the applications we examined contained simple loops around disjoint approximate function calls. For example, the `sobel` application used in our evaluation consists of a stencil computation that resembles the pixel filter above.

### 3. Architecture

This work takes advantage of an emerging class of heterogeneous computing devices, Programmable System-on-Chips (PSoCs). These devices combine a hard processor core with programmable logic in the same package and support low-latency communication between the two. SNNAP’s architecture is informed by the challenges and opportunities involved in working with PSoCs. A good design for PSoCs must exploit the low-latency link with the CPU while simultaneously maximizing throughput and energy efficiency.

**Requirements** The requirements for an effective NPU design in a PSoC are as follows:

- The NPU must be configurable to support different neural network architectures.
- It must operate independently of the CPU to allow the CPU to sleep and conserve energy.
- It must use the FPGA fabric efficiently to minimize its energy consumption.
- It must support low-latency invocations to provide benefit to code with limited approximate region coverage.
- It must also support high-throughput in codes that support batched concurrent invocation.

The rest of this section describes, at a high level, our techniques for exploiting the particular characteristics of the PSoC to meet these requirements.

#### 3.1. Design Overview

**A systolic array implementation of neural networks.** We choose to base our microarchitectural design on the concept of *systolic arrays*. Systolic arrays excel at exploiting regular data-parallelism, which is abundant in neural networks [12]. Furthermore, systolic arrays are amenable to efficient implementation on modern FPGAs for two reasons. First, systolic arrays are highly pipelined and FPGAs have high ratio of registers to combinational logic resources, which facilitates efficient pipelining. Second, most of a systolic array’s datapath can be contained within the dedicated Digital Signal Processing (DSP) silicon found in FPGAs. We leverage these resources to realize an efficient systolic array based FPGA design, which in turn motivates the throughput-oriented interface to SNNAP.

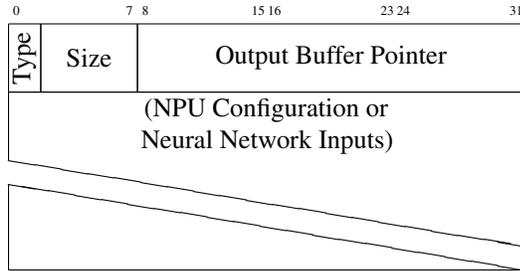


Figure 1: NPU invocation record.

**The Neural Processing Unit.** The core of SNNAP is the *neural processing unit* (NPU). The NPU is composed of four main components: an array of processing elements (PEs), weight memories, a sigmoid unit (SU), and a scheduler. During the evaluation of a neural network, the computation of the neurons are multiplexed onto the PEs and the SU by the scheduler.

The PEs constitute the computational substrate of the systolic array, and are implemented on the FPGA’s hard DSP block. The weight memories are distributed across the PEs and store the weights generated as part of the neural network training process. The sigmoid unit uses a lookup-table implementation of a non-linear sigmoid activation function.

Different neural networks have different weights, topologies and can use different variants of the sigmoid function. As part of the NPU configuration process, the weights are loaded into the weight memories, the neural network schedules are loaded into the scheduler, and the sigmoid activation function values can be loaded into the lookup-table in the SU.

Section 4 gives more details on the implementation of the NPU.

**Hardware interface.** There are three components to the CPU–SNNAP interface: a memory-mapped register, the ARM `SEV/WFE` instruction signaling mechanism, and the ARM Accelerator Coherency Port.

SNNAP communicates with the CPU primarily through cache-coherent shared memory. This interface must support two operations: reconfiguration (`nn_configure`) and neural network execution (`nn_send` and `nn_receive`). These operations are performed by writing a descriptor to memory and signaling SNNAP using the ARM `SEV` instruction; the accelerator reads the descriptor, performs the operations it contains, and signals the CPU.

Before SNNAP can be used, its descriptor base address must be set. Our current prototype uses a fixed base address set at design time, but it could also be made configurable by writing into a memory-mapped register. When SNNAP is signaled, it starts reading from this address. Descriptor addresses must be 256-byte aligned; thus, only 24 bits of this register are significant.

Figure 1 shows the format of a descriptor record. The Type field denotes the kind of descriptor: 0 for configuration and 1 for invocation. The Size field denotes the length of the record

in 32-bit words. The Output Pointer points to a 256-byte aligned buffer to be used for neural-network outputs.

To perform a computation on SNNAP, the CPU builds a buffer of invocations and passes the pointer to SNNAP, which then reads the buffer and executes the invocations. When the neural-network execution is complete, SNNAP writes the results to the output buffer as they are computed and signals the processor when they are all done.

### 3.2. Zynq PSoC Specifics

This section describes the particular challenges faced when designing SNNAP for a real PSoC device available today. Our evaluation uses the Xilinx Zynq-7020 on the ZC702 evaluation platform [50]. The features described here are not unique to Xilinx’s implementation; other manufacturers produce similar devices [2].

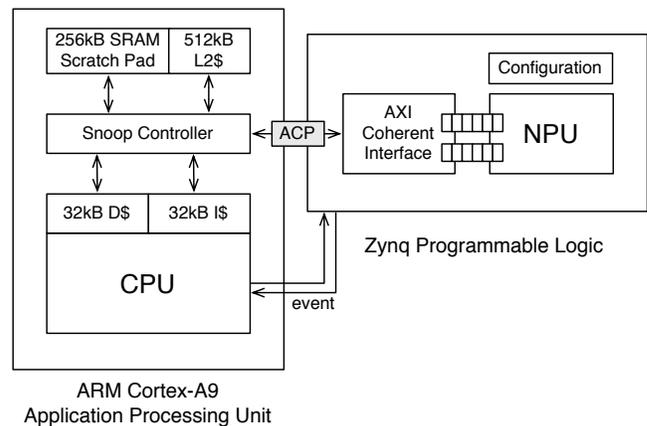


Figure 2: SNNAP system overview.

Figure 2 shows an overview of the SNNAP system, including the NPU.

**3.2.1. The ARM processors and FPGA fabric.** The Zynq includes dual ARM Cortex-A9 cores which support the ARM version 7 ISA. Each core has a 32KB instruction cache and a 32KB data cache, and the two cores share a 512KB L2 cache. The address space is backed by a DRAM controller, driving 1GB of DDR3 DRAM on the board, as well as a 256 KB scratchpad SRAM. The Zynq SoC includes a number of hard peripherals including interfaces for Gigabit Ethernet, USB, SPI, I2C, CAN, and an analog-to-digital converter.

The SoC includes a fairly standard programmable logic fabric: 6-input lookup tables, programmable interconnect, 36 Kb SRAMs (referred to as BRAMs), and hard multiply-accumulate units (referred to as DSP blocks), supporting  $25 \times 18$ -bit multiplies with a 48-bit accumulator.

Because the ARM subsystem is hard logic, its maximum clock rate is relatively fast. The device we used tops out at 666 MHz, although other devices in the Zynq family support clock rates up to 1 GHz. While the maximum clock rate obtainable

when using the FPGA fabric is dependent on the design, it is limited by the maximum clock rate of the SRAMs (388 MHz) and MAC units (464 MHz); once the delay due to routing is taken into account, we have found it practical to target clock rates in the 200–250 MHz range. The accelerator exploits parallelism in the neural network to overcome this difference.

**3.2.2. CPU–FPGA communication.** While PSoCs hold the promise of low-latency, high-bandwidth communication between the CPU and FPGA, the reality is more complicated. We need a communication design that permits throughput-oriented, asynchronous neural-network invocations without sacrificing latency. The Zynq device supports five different models of communication between the CPU and the FPGA [52]: direct synchronization signaling, low throughput CPU programmed I/Os, medium-throughput General Purpose (GP) I/Os, a high-throughput Accelerator Coherency Port (ACP) and very high-throughput High Performance (HP) ports.

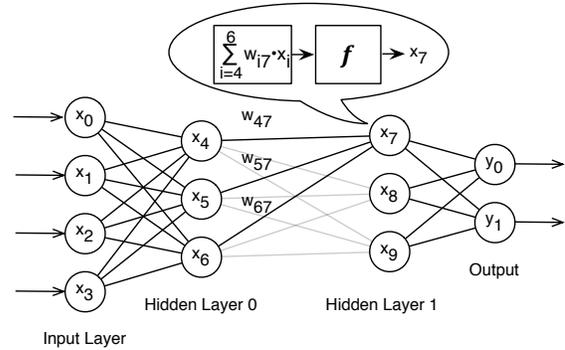
**Synchronization.** The ARM v7 ISA contains two instructions for synchronizing with accelerators. The ARM and the FPGA are connected by two unidirectional event lines `eventi` and `evento`, used for direct synchronization. The `SEV` instruction causes a the `evento` in the FPGA fabric to toggle; the `WFE` instruction causes the processor to sleep in a low-power state until the FPGA toggles the `eventi` signal. These operations have lower latency (5 CPU cycles) than any of the other communication methods and are used in our design for synchronization purposes.

**CPU programmed I/Os.** Up to 64 bits connected to the FPGA fabric can be accessed by the ARM processors as memory-mapped registers. These offer a relatively medium-latency interface with the FPGA (138 CPU cycle roundtrip latency), but are also low-bandwidth, since transitions must be executed by the CPU a word at a time.

**General Purpose (GP) I/Os.** The ARM interconnect includes two 32-bit AXI bus GP interfaces to the FPGA fabric, which can be used to implement memory-mapped registers or support DMA transfers. These interfaces are easy to use and are relatively low-latency (114 CPU cycle roundtrip latency) but only support moderate bandwidth. We found the GP I/O interface useful to implement memory-mapped registers for pointer passing.

**High Performance (HP) I/Os** The ARM interconnect includes four 64-bit AXI slave interfaces connected directly to the memory system. This allows the FPGA to issue reads and writes directly to the ARM SRAM and DRAM controller. However, these operations are not coherent with the processor’s caches. The CPU must flush the data from the cache to make it visible to the FPGA via main memory. The HP interface is appropriate for large data movements to and from external DRAM, which is currently not adequate for the data granularity we are operating on.

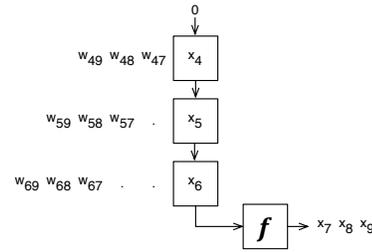
**Accelerator Coherency Port (ACP).** The FPGA can access the ARM on-chip memory system through the 64-bit Accelerator Coherency Port AXI slave interface. This port allows



(a) An MLP neural network.

$$\begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = f \left( \begin{pmatrix} w_{47} & w_{57} & w_{67} \\ w_{48} & w_{58} & w_{68} \\ w_{49} & w_{59} & w_{69} \end{pmatrix} \cdot \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} \right)$$

(b) Matrix representation of hidden layer evaluation.



(c) Systolic algorithm on one-dimensional systolic array.

**Figure 3: Implementing multi-layer perceptron neural networks with systolic arrays.**

the FPGA to make read and write requests directly to the processors’ Snoop Control Unit, allows the FPGA to read data directly out of the processor caches with high bandwidth and low latency. We found this communication interface to be the ideal candidate given the data sizes we were moving (4B-4kB) at a time. We ended up designing a custom AXI master for the ACP interface to optimize our design’s maximum frequency and to reduce roundtrip communication latency down to 93 CPU cycles.

Since SNNAP needs to operate independently from the CPU and to support high-throughput, low-latency coherent transfers, we choose the Accelerator Coherency Port to move data and `SEV/WFE` instructions to synchronize with the CPU.

## 4. Hardware Design

This section describes SNNAP’s microarchitecture. We first describe the use of systolic arrays to evaluate a multi-layer perceptron (MLP) neural network. We then discuss our specific FPGA implementation.

#### 4.1. Multi-Layer Perceptrons With Systolic Arrays

A *multi-layer perceptron* is a type of neural network that consists of a set of nodes, arranged in *layers*, connected in a directed graph. Each layer is fully connected to the next. Each edge has a weight, and each neuron is a computation unit that computes the weighted sum of its inputs and applies a non-linear *activation function*. This function is most commonly a sigmoid function.

Figure 3a depicts an MLP with two hidden layers. The computation of one of the neurons in the second hidden layer is highlighted: the neuron multiplies the values of the source neurons with the weights, sums them, and applies the nonlinear activation function  $f$  to the result. The activation function yields the output value to be sent to the destination neuron.

The evaluation of an MLP neural network can be represented by a series of matrix–vector multiplications interleaved with non-linear activation functions. Figure 3b shows this approach applied to the hidden layers of Figure 3a. We can schedule a systolic algorithm for computing this matrix-by-vector multiplication onto a 1-dimensional systolic array as shown in Figure 3c. When computing a layer, the vector elements  $x_i$  are loaded into each cell in the array as elements of the matrix  $w_{ji}$  trickle in. Each cell is a computational element that performs a multiplication  $x_i \cdot w_{ji}$ , adds it to the sum of products produced by the upstream cell above it, and sends the result to the downstream cell below it. The activation function is then applied to the output vector produced by the systolic array, completing the layer computation.

Systolic arrays can be efficiently implemented using the hard DSP slices that are common in modern FPGAs. Our Zynq unit incorporates 220 DSP slices in its programmable logic [52]. DSP slices offer pipelined fixed-point multiply-and-add functionality and provide a hard-wired data bus for fast aggregation of partial sums on a single column of DSP slices. As a result, a one-dimensional systolic array can be contained entirely in a single hard logic unit to provide higher performance and lower power [51].

#### 4.2. SNNAP’s Systolic Array Design

The core of SNNAP is the Neural Processing Unit (NPU), which implements a complete configurable neural network. The NPU design consists of a series of Processing Elements (PEs), as illustrated in Figure 4a. Each PE corresponds to a systolic array cell. As shown in Figure 4b, a PE consists of a multiply-and-add block implemented on a DSP slice.

The input elements are loaded every cycle via the input bus into each PE as dictated by the systolic algorithm schedule. Weights, on the other hand, are statically partitioned among the PEs in weight memories. Each weight is stored in the weight memory associated with the PE that will compute the corresponding neuron value. These weight memories are implemented on an FPGA Block RAM (BRAM).

The NPU architecture can support an arbitrary number of

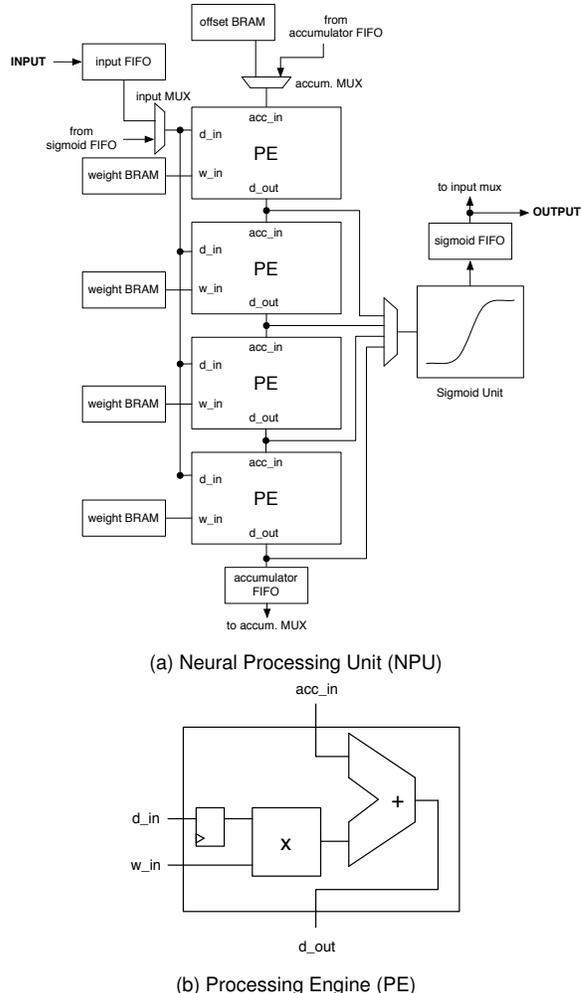


Figure 4: SNNAP’s systolic array design.

PEs; however, adding more PEs in the design causes the input bus length and fan-out to increase, thus increasing complexity and decreasing maximum operating frequency.

**Sigmoid Unit.** The NPU design uses a 3-stage pipelined sigmoid unit that applies a non-linear activation function on each outputs produced by the PEs. The Sigmoid Unit is implemented using a BRAM-based lookup-table (LUT) plus some logic for special cases. We found that using a 2048-entry LUT provided adequate accuracy (0.011% normalized RMSE over the input range) when combined with a  $x = y$  linear approximation on small input values. Due to the multitude of neural network configurations, we support three activation functions: (1) the sigmoid activation function, (2) the hyperbolic tangent function and (3) a linear activation function, sometimes used on the output layer. The configuration determines which activation function is used for each layer.

We found that one Sigmoid Unit was sufficient for our NPU design. Out of the 6 application benchmarks we used in our evaluation, only the FFT benchmark schedule experienced contention for the Sigmoid Unit from two PEs, thus introduc-

ing a one-cycle bubble in the schedule and increasing the FFT neural network computation latency by 4.8%.

**Flexible NN size and topology.** While abstract neural networks can use any number of neurons, the NPU design must have a fixed size. The NPU executes neural networks of different sizes by time-multiplexing the neural network computation onto the PEs. The systolic algorithm breaks down each MLP layer into multiple sequences of sum-of-product operations. Computing a MLP layer with  $n$  input neurons and  $m$  output neurons requires  $n$  PEs, each performing  $m$  multiplications.

If a layer has fewer than  $n$  inputs (where  $n$  is the number of inputs), the outputs of the unused PEs are ignored. On the other hand, evaluating a neural network with more than  $n$  input neurons requires time-multiplexing the PEs. We include an *accumulator FIFO* in our design to store the temporary sums produced by the PEs during this process. The PEs are multiplexed in order: first, the partial sums of the first  $n$  input neurons are computed and stored in the accumulator FIFO; then the PEs are set to compute the next  $n$  input neurons and the partial sums are streamed through. This process repeats until the last input neuron is mapped to a PE; at that point the completed sum is directed to the sigmoid unit.

A similar process is performed to evaluate neural networks with hidden layers. In this case, the outputs of the sigmoid unit must be buffered until the evaluation of the layer is complete; then they can be used as inputs to the next layer. This is the purpose of the *sigmoid FIFO*. When evaluating the final layer in a neural network, the outputs coming from the sigmoid unit are sent directly to the memory interface to be written back to the CPU’s memory.

The fixed sizes of the sigmoid and accumulator FIFOs limit the maximum layer width of the neural networks that SNNAP can execute. This limit can be alleviated by allocating more BRAM space for each FIFO.

**Control.** The topology of a neural network can be mapped to a static schedule for the NPU given the number of PEs in the NPU. The process is computationally inexpensive and can be computed on-the-fly during a `nn_configure` call. Note this does not involve reconfiguring the FPGA itself.

The schedule is represented as vertical microcode and stored in a BRAM. Outputs of this BRAM are used as control inputs to the PEs, Sigmoid Unit, FIFOs and weight memories. When the NPU starts evaluating a neural network, it iterates through the schedule memory in order without stopping.

**Numeric representation.** While many approximate applications use floating-point data types, floating point computations are difficult to implement efficiently in an FPGA. To enable the use of an efficient fixed-point numeric representation inside the SNNAP design, we bound the range of possible neuron weights in a neural network configuration.

The NPU design optionally converts from floating-point to fixed-point at its input and output ports and uses a fixed-point representation internally. We selected a 16-bit signed fixed-

point representation with 7 fractional bits to make efficient use of the ARM’s byte-oriented memory interface for applications that provide fixed-point inputs directly. This representation fits within the  $18 \times 25$  DSP slice multiplier blocks. The DSP slices also include a wide 48-bit fixed-point adder that helps avoid overflows on long summation chains.

## 5. Evaluation

### 5.1. Experimental setup

**Applications.** Table 1 shows the applications measured in this evaluation, which are the same benchmarks used by Esmaeilzadeh et al. [20]. We offloaded one approximate region from each application to SNNAP using the neural acceleration transformation process. We include a hypothetical “Amdahl speedup limit” computed by subtracting the measured runtime of the kernel to be accelerated from the overall benchmark runtime.

**Target Platform.** We evaluate the performance, power and energy efficiency of SNNAP running against software on the ZYNQ ZC702 evaluation platform described in Table 2. The ZYNQ processor integrates a mobile-grade ARM Cortex-A9 and a Xilinx FPGA fabric on a single TSMC 28nm die.

**Software environment.** We compiled our benchmarks using GCC 4.7.2 targeting the ARM Cortex-A9 architecture, with the `-O3` flag. We ran the benchmarks directly on the processor, with no OS.

**Monitoring performance and power.** In order to count dynamic instructions and CPU cycles on the Cortex-A9 core, we use the event counters in the ARM’s architectural performance monitoring unit, combined with performance counters implemented in the FPGA.

The ZYNQ ZC702 platform uses Texas Instruments UCD9240 power supply controllers, which allow us to measure voltage and current usage on each of the board’s power planes multiple times a second. This allows us to track power usage for the different sub-systems (e.g. CPU, FPGA, DRAM).

**NPU configuration.** We evaluated our design with an 8-PE NPU, running at 222MHz, maintaining a 1/3 integer ratio with CPU’s 666MHz frequency. The number of PEs was fixed at 8 arbitrarily in order to support the construction of a simpler design; future work could evaluate the benefits of different PE counts.

### 5.2. Performance and Energy

Transforming a program to use SNNAP can improve performance and energy efficiency. Here, we describe empirical measurements of those benefits on our test hardware.

**Performance.** Figure 5a shows the application speedup when an 8-PE NPU is used to execute each benchmark’s target region, while the rest of the application runs on the CPU,

Application	Description	Error Metric	NN Topology	Error	Amdahl Speedup ( $\times$ )
fft	radix-2 Cooley-Tukey FFT	mean error	1-4-4-2	0.1%	3.92
inversek2j	inverse kinematics for 2-joint arm	mean error	2-8-2	1.32%	> 100
jmeint	triangle intersection detection	miss rate	18-32-8-2	20.47%	99.65
jpeg	lossy image compression	image diff	64-16-64	1.93%	2.23
kmeans	$k$ -means clustering	image diff	6-8-4-1	2.55%	1.47
sobel	edge detection	image diff	9-8-1	8.57%	15.65

**Table 1: Applications used in our evaluation. The “NN topology” column shows the number of neurons in each MLP layer. “Amdahl Speedup” is the hypothetical speedup for a system where the SNNAP invocation is instantaneous.**

Zynq SoC		Cortex-A9		NPU	
Technology	28nm TSMC	L1 Cache Size	32kB I\$, 32kB D\$	Number of PEs	8
Processing	2-core Cortex-A9	L2 Cache Size	512kB	Weight Memory	1024 $\times$ 16-bit
FPGA	Artix-7	Scratch-Pad	256kB SRAM	Sigmoid LUT	2048 $\times$ 16-bit
FPGA Capacity	53KLUTs, 106K Flip-Flops	Interface Port	AXI 64-bit ACP	Accumulator FIFO	1024 $\times$ 48-bit
Peak Frequencies	667MHz A9, 222MHz FPGA	Interface Latency	93 cycles roundtrip	Sigmoid FIFO	1024 $\times$ 16-bit
DRAM	1GB DDR3-533MHz			DSP Unit	16 $\times$ 16-bit multiply, 48-bit add

**Table 2: Microarchitectural parameters for the Zynq platform, CPU, FPGA and NPU.**

compared to running the whole benchmark on the CPU. On average, we see a speedup of  $1.77\times$ .

Among the benchmarks, *inversek2j* has the highest speedup ( $17.95\times$ ) since the bulk of the application can be executed on the NPU, and the region to be accelerated includes many trigonometric functions, which the NPU can approximate using only a small neural network. Conversely, *kmeans* sees a 40% slowdown, mostly due to the fact that the target region is very small (only a handful of arithmetic operations), compared to the size of the neural network required to achieve reasonable error rates.

**Energy.** Figure 5b shows the energy savings for each benchmark. The baseline is the energy consumed by executing the benchmark solely on the CPU. This comparison shows the actual energy savings we measured on our board, including all the components.

We find that all the benchmarks that provide a speedup are able to save energy. Again, *inversek2j* does the best ( $14.12\times$ ), and *kmeans* does the worst ( $0.41\times$ ). *sobel* has the smallest actual speedup, and saves a small amount of energy ( $1.08\times$ ). These energy savings can be understood by looking at the power draw of the CPU and FPGA: when the CPU is offloading computation to the NPU, it sleeps, lowering its power draw slightly. But the power required to perform computation on the NPU on the FPGA is larger than this savings. Thus, a speedup is necessary to obtain a energy savings.

### 5.3. Characterization

This section supplements our main energy and performance results with secondary measurements that put the primary results in context and help justify our design decisions.

**Energy savings breakdown.** While we observed an energy savings on average for our benchmarks, the evaluation board we used is not optimized for low-power operation. In Figure 6, we provide a more detailed breakdown of the energy savings we measured to help predict the benefit for new, power-optimized designs.

The first comparison is intended to provide a conservative estimate of the potential benefit to a mobile SoC designer who is considering including an FPGA fabric in her design. This comparison includes only the power drawn by the core logic and SRAMs of the CPU and FPGA, with no DRAM or peripherals included. The baseline includes only the core logic of the CPU, with no FPGA power. The bars show the energy benefit when running on the NPU, including both core logic and memory power planes for the CPU and the FPGA fabric. This comparison shows similar results to the whole-board speedup numbers, but with a lower magnitude. On average, we see a  $1.31\times$  energy savings.

The second comparison is intended to show the power benefit from using an NPU on a chip that already has an FPGA fabric. This baseline includes all the power supplies that connect to the Zynq chip and its associated DRAM, including the FPGA core voltage, but the FPGA is left unconfigured during software-only execution. We omit the 3.3 volt supply to avoid making an unfair comparison: on our board this supply powers wasteful components (e.g., high-current LEDs) that would not be included in a design optimized for power. The bars show the energy benefit when the FPGA is programmed and benchmarks are accelerated on the NPU.

Again, all the benchmarks that provide a speedup are again able to save energy, but this time the benefit is larger ( $1.64\times$  on average). This is due to the power draw of components common to both the baseline and accelerated cases. The additional power drawn by the FPGA during acceleration is smaller rela-

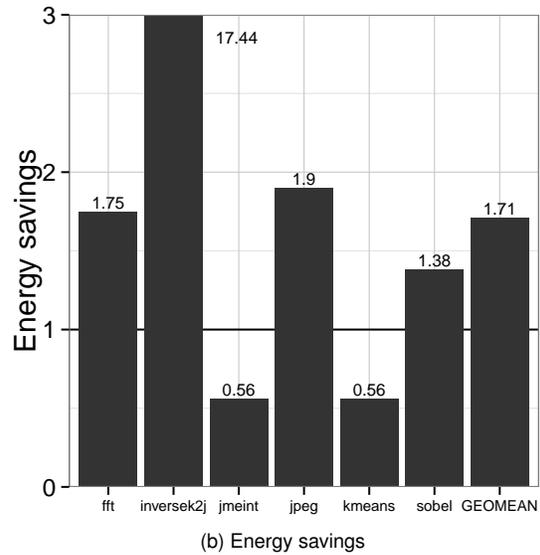
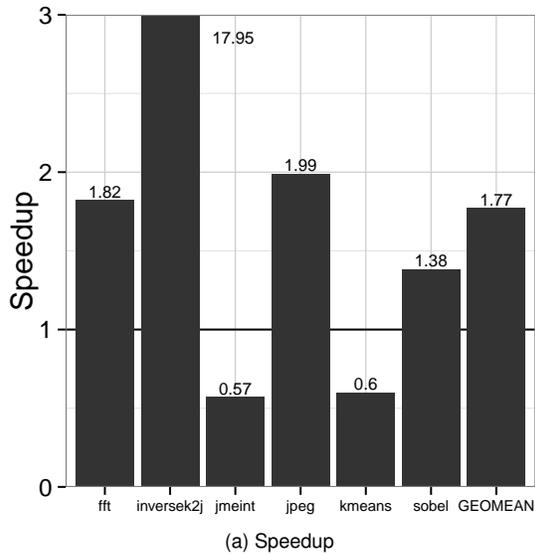


Figure 5: Performance and energy benefit of SNNAP acceleration over an all-CPU baseline execution of each benchmark.

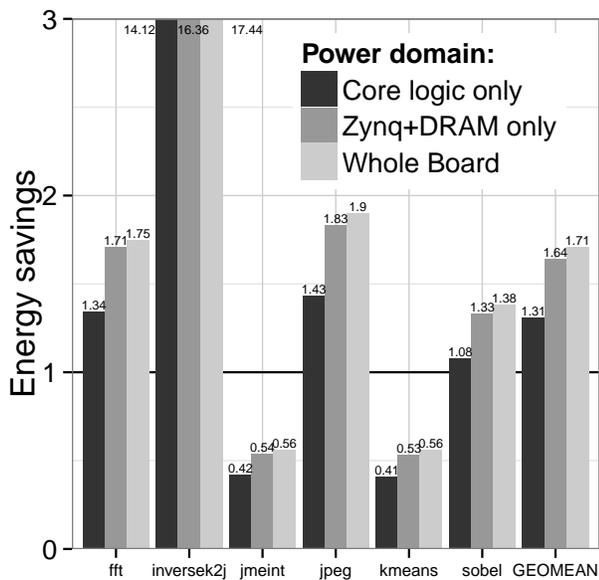


Figure 6: Energy savings breakdown.

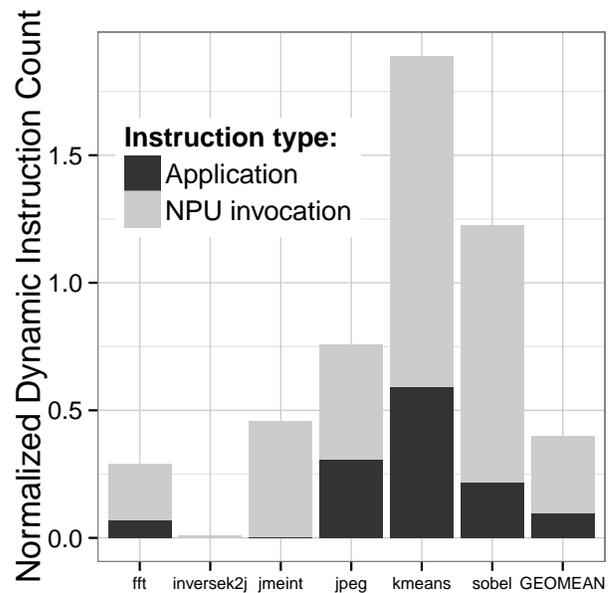


Figure 7: Number of dynamic instructions executed with NPU acceleration normalized to the original program.

tive to the total power draw of the system, making the relative energy benefit of acceleration larger.

**Dynamic instruction subsumption.** Figure 7 shows the number of dynamic instruction of each accelerated benchmark normalized to the instruction count of the CPU-only version. Instructions are divided into those executed by both the original and accelerated programs, and those used to communicate with the NPU.

The reduction in instruction count is closely related to the speedup obtained from NPU acceleration. The fastest benchmark, *inversek2j*, is an ideal application for the NPU. The region to be accelerated accounts for 99 percent of the original dynamic instruction count—each invocation executes many

trigonometric functions—and the inputs and outputs are small (two single-precision floating point numbers each). In contrast, *kmeans* is a bad fit for NPU-based acceleration. Its target region accounts for only 40 percent of the original dynamic instruction count, and the actual region of code to be accelerated (a Euclidean distance calculation) is can be executed with a handful of instructions but requires the movement of seven single-precision floating point numbers. The number of instructions required just to invoke one instance of the computation on the NPU is more than the instruction count to perform the computation on the CPU.

However, the dynamic instruction count does not tell the

whole story. Since the processor sleeps while the accelerated region is executing on the NPU, the instruction count does not reflect the latency of the NPU computation. This is illustrated by `jmeint`, where the target region accounts for nearly all the original dynamic instructions, but the neural network required for the computation is complex and adds significant latency to the computation while the CPU is sleeping.

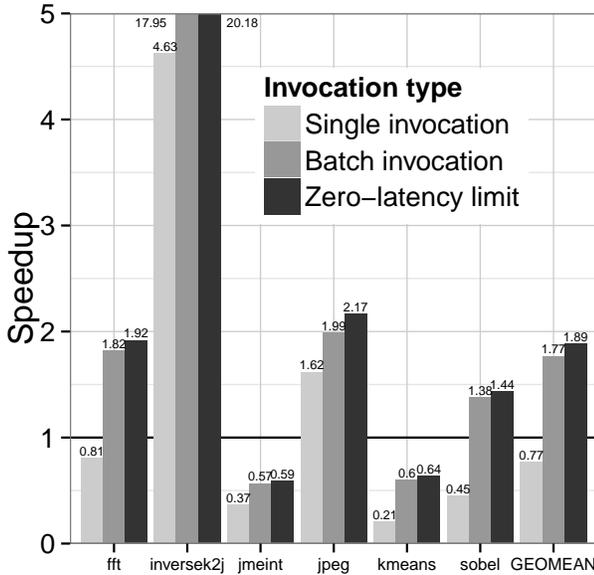


Figure 8: Impact of batching on speedup.

**Impact of batching.** Figure 8 compares the performance of batched SNNAP invocations, single invocations, and an estimate of the speedup if there were no communication latency between the CPU and the NPU.

Overlapped, batched invocations are required to obtain a speedup in all but two cases due to the latency in communicating with the NPU. `inversek2j` and `jpeg` see a speedup since their regions to be accelerated are large enough to overlap the latency of each invocation. As discussed previously, the accelerated region in `inversek2j` includes many expensive trigonometric functions. The accelerated region in `jpeg` is similarly computationally heavy.

Comparing with the zero-latency estimate, we find that batch invocations are effective at amortizing this latency across many invocations. The 32-invocation batch size we have chosen gets us to within 7% of the zero-latency case.

**Design statistics.** The SNNAP design uses less than 5% of the FPGA resources on the Zynq part we used, as shown in table 3. The small design footprint helps reduce the power and indicates that our design could be replicated on a significantly smaller FPGA die.

We managed to close timing of our design at 222MHz, maintaining a 1/3 integer ratio with CPU’s 666MHz frequency.

Logic Utilization	Used	Available	Util
Occupied Slices	625	13300	4%
Slice Registers	2055	106400	2%
Slice LUTs	1650	53200	3%
RAMB18E1	13	280	4%
RAMB36E1	4	140	2%
DSP48E1	8	220	3%

Table 3: Post-place-and-route FPGA utilization.

**Output quality degradation.** We measure the application output quality degradation on the benchmarks running on SNNAP using application-specific error metrics as is standard in the approximate computing literature [19,20,40,43]. Table 1 lists the error metrics.

We observe less than 10% application output error for all of the benchmarks except for `jmeint`. For that benchmark, we were unable to reproduce the low error rates from Esmaeilzadeh et al. [20] using the same neural-network topology, even when executing the neural network in software.

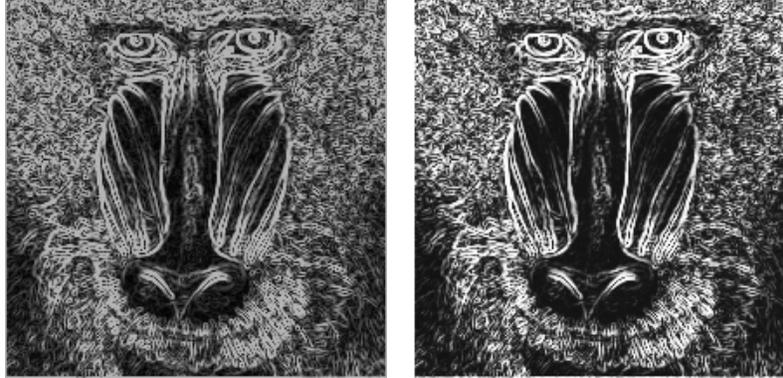
Among the remaining applications, the highest output error occurs in `sobel`. The output image has a 8.57% mean absolute pixel error with respect to a precise execution. To put this error in context, Figure 9 shows the output from the original and SNNAP-accelerated executions of the benchmark. Qualitatively, the program still produces reasonable results.

## 6. Related Work

Our design builds on related work in the broad areas of approximate computing, acceleration, and neural networks.

**Approximate Computing** A wide variety of applications can be considered *approximate*: occasional errors during execution do not obstruct the usefulness of the program’s output. Recent work has proposed to exploit this inherent resiliency to trade off output quality to improve performance or energy consumption using software [3,4,28,33,34,43] or hardware [8,15,19,20,30,31,35,40] techniques. SNNAP represents the first work (to our knowledge) to exploit this trade-off using on-chip programmable logic to realize these benefits in the near term. FPGA-based acceleration using SNNAP offers efficiency benefits that complement software approximation, which is limited by the overheads of general-purpose CPU execution, and custom approximate hardware, which cannot be realized on today’s chips.

**Neural Networks as Accelerators** Previous work has recognized the potential for hardware neural networks to act as accelerators for approximate programs, either with automatic compilation [20] or direct manual configuration [5,10,46]. This work has typically assumed special-purpose neural-network hardware; SNNAP represents an opportunity to realize these benefits in the near term.



(a) Precise Output

(b) Approximate Output

Figure 9: Output of `sobel` for a 220x220 pixel image.

**Hardware Support for Neural Networks** There is an extensive body of work on hardware implementation of neural networks both in digital [17, 37, 53] and analog [6, 29, 42, 45] domains. Recent work has proposed higher-level abstractions for implementation of neural networks [27]. Other work has examined fault-tolerant hardware neural networks [26, 46]. In particular, Temam [46] uses datasets from the UCI machine learning repository [22] to explore fault tolerance of a hardware neural network design. There is also significant prior effort on FPGA implementations of neural networks ([53] contains a comprehensive survey).

Even though our work involves efficient implementation of neural networks on FPGAs, our contribution focuses on providing mechanisms that automatically leverages that implementation for approximate acceleration without engaging the programmers in hardware design.

Recent work explored using GPUs for large-scale neural networks [14] and showed significant performance improvements over general purpose processors. Indeed, GPUs are highly parallel (as our systolic array is) and therefore a good match for the kind of computation involved in neural networks. The neural acceleration transformation for general purpose approximation could use GPUs as the backend, however there are likely at least two issues: (1) the invocation cost of a neural network is much higher on GPUs [32], so code regions have to be either very coarse or very parallel, limiting its applicability and code coverage; and (2) GPUs are much more power hungry. We consider exploring these issues out of the scope of this paper.

**FPGAs as Accelerators** This work also relates to work on configurable computing, synthesis, specialization, and acceleration that focuses on compiling traditional, imperative code for efficient hardware structures. One key research direction seeks to synthesize efficient circuits or configure FPGAs to accelerate general-purpose code [13, 21, 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 [23, 24]. 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 by converting regions of code to neural networks. Furthermore, automatic hardware synthesis results in suboptimal hardware design [1, 16], especially on FPGAs due to their low frequency and lack of information about dynamic behavior (e.g., memory dependences) often limit the quality of the output design.

In contrast, we manually design our neural accelerator and optimize it as a library component that is abstracted away from the programmer and used automatically.

## 7. Conclusion

We presented SNNAP, a system that enables the use of programmable logic to accelerate general purpose programs without requiring hardware design. SNNAP leverages prior work on using neural networks to emulate regions of approximable code. Since neural networks are amenable to efficient hardware implementations, this leads to better performance and energy efficiency.

We implemented SNNAP on the Zynq system-on-chip, which contains a significant amount of programmable logic on chip. We developed a systolic-array-based multi-layer perceptron design, which uses a small fraction of the available on-chip programmable logic but yields  $1.7\times$  speedup and energy savings on average.

This exercise demonstrates that approximate computing techniques can enable effective use of programmable logic for general purpose acceleration without involving custom logic design, nor direct high-level synthesis or frequent FPGA reconfiguration.

## 8. Acknowledgments

This work was supported by the Qualcomm Innovation Fellowship, the NSF, the NSERC, the Google Ph.D. Fellowship

and the Weil Family. We thank Eric Chung from Microsoft Research for his help on building a custom AXI mastering interface.

## References

- [1] “A parameterized graph-based framework for high-level test synthesis,” *Integration, the VLSI Journal*, vol. 39, no. 4, pp. 363–381, 2006.
- [2] Altera Corporation, “Altera SoCs.” Available: <http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html>
- [3] J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe, “PetaBricks: a language and compiler for algorithmic choice,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.
- [4] W. Baek and T. M. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.
- [5] B. Belhadj, A. Joubert, Z. Li, R. Heliot, and O. Temam, “Continuous real-world inputs can open up alternative accelerator designs,” in *International Symposium on Computer Architecture (ISCA)*, 2013, pp. 1–12.
- [6] 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, no. 12, pp. 2017–2025, December 1991.
- [7] M. Carbin, S. Misailovic, and M. C. Rinard, “Verifying quantitative reliability for programs that execute on unreliable hardware,” in *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2013, pp. 33–52.
- [8] 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 (PCMO) technology,” in *Design, Automation and Test in Europe (DATE)*, 2006, pp. 1110–1115.
- [9] I. J. Chang, D. Mohapatra, and K. Roy, “A priority-based 6t/8t hybrid sram architecture for aggressive voltage scaling in video applications,” *IEEE Trans. Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, 2011.
- [10] 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 *IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 36–45.
- [11] E. S. Chung, J. C. Hoe, and K. Mai, “CoRAM: An in-fabric memory architecture for fpga-based computing,” in *FPGA*, 2011.
- [12] J.-H. Chung, H. Yoon, and S. R. Maeng, “A systolic array exploiting the inherent parallelisms of artificial neural networks,” vol. 33, no. 3. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., May 1992, pp. 145–159. Available: [http://dx.doi.org/10.1016/0165-6074\(92\)90017-2](http://dx.doi.org/10.1016/0165-6074(92)90017-2)
- [13] 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 *International Symposium on Microarchitecture (MICRO)*, 2004, pp. 30–40.
- [14] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, “Deep learning with cots hpc systems,” 2013.
- [15] M. de Kruijff, S. Nomura, and K. Sankaralingam, “Relax: An architectural framework for software recovery of hardware faults,” in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 497–508.
- [16] G. de Micheli, Ed., *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [17] H. Esmaeilzadeh, P. Saeedi, B. Araabi, C. Lucas, and S. Fakhraie, “Neural network stream processing core (NnSP) for embedded systems,” in *International Symposium on Circuits and Systems (ISCAS)*, 2006, pp. 2773–2776.
- [18] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *International Symposium on Computer Architecture (ISCA)*, 2011.
- [19] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 301–312.
- [20] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *International Symposium on Microarchitecture (MICRO)*, 2012, pp. 449–460.
- [21] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, “Bridging the computation gap between programmable processors and hardwired accelerators,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 313–322.
- [22] A. Frank and A. Asuncion, “UCI machine learning repository,” 2010. Available: <http://archive.ics.uci.edu/ml>
- [23] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, “Dynamically specialized datapaths for energy efficient computing,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 503–514.
- [24] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, “Bundled execution of recurring traces for energy-efficient general purpose processing,” in *International Symposium on Microarchitecture (MICRO)*, 2011, pp. 12–23.
- [25] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, “Toward dark silicon in servers,” *IEEE Micro*, vol. 31, no. 4, pp. 6–15, July–Aug. 2011.
- [26] A. Hashmi, H. Berry, O. Temam, and M. H. Lipasti, “Automatic abstraction and fault tolerance in cortical microarchitectures,” in *International Symposium on Computer Architecture (ISCA)*, 2011, pp. 1–10.
- [27] A. Hashmi, A. Nere, J. J. Thomas, and M. Lipasti, “A case for neuro-morphic ISAs,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 145–158.
- [28] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, “Dynamic knobs for responsive power-aware computing,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [29] A. Joubert, B. Belhadj, O. Temam, and R. Heliot, “Hardware spiking neurons design: Analog or digital?” in *International Joint Conference on Neural Networks (IJCNN)*, 2012, pp. 1–7.
- [30] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, “ERSA: Error resilient system architecture for probabilistic applications,” in *DATE*, 2010.
- [31] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, “Flicker: Saving dram refresh-power through critical data partitioning,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 213–224.
- [32] D. Lustig and M. Martonosi, “Reducing gpu offload latency via fine-grained cpu-gpu synchronization,” in *HPCA*, 2013.
- [33] S. Misailovic, D. Kim, and M. Rinard, “Parallelizing sequential programs with statistical accuracy tests,” MIT, Tech. Rep. MIT-CSAIL-TR-2010-038, Aug. 2010.
- [34] S. Misailovic, D. M. Roy, and M. C. Rinard, “Probabilistically accurate program transformations,” in *Static Analysis Symposium (SAS)*, 2011.
- [35] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, “Scalable stochastic processors,” in *Design, Automation and Test in Europe (DATE)*, 2010, pp. 335–338.
- [36] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, “Triggered instructions: A control paradigm for spatially-programmed architectures,” in *International Symposium on Computer Architecture (ISCA)*, 2013.
- [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 *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2008, pp. 261–261.
- [39] R. Razdan and M. D. Smith, “A high-performance microarchitecture with hardware-programmable functional units,” in *International Symposium on Microarchitecture (MICRO)*, 1994, pp. 172–180.
- [40] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 164–174.
- [41] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, “Approximate storage in solid-state memories,” in *International Symposium on Microarchitecture (MICRO)*, 2013.
- [42] J. Schemmel, J. Fieres, and K. Meier, “Wafer-scale integration of analog neural networks,” in *International Joint Conference on Neural Networks (IJCNN)*, 2008, pp. 431–438.
- [43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, “Managing performance vs. accuracy trade-offs with loop perforation,” in *Foundations of Software Engineering (FSE)*, 2011.

- [44] S. Sirowy and A. Forin, "Where's the beef? why fpgas are so fast," Microsoft Research, Tech. Rep. MSR-TR-2008-130, Sep. 2008.
- [45] S. Tam, B. Gupta, H. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in *Systems, Man, and Cybernetics (SMC)*, 1990, pp. 701–703.
- [46] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 356–367.
- [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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 205–218.
- [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 *International Symposium on Microarchitecture (MICRO)*, 2011, pp. 163–174.
- [49] M. Weber, M. Putic, H. Zhang, J. Lach, and J. Huang, "Balancing adder for error tolerant applications," in *International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 3038–3041.
- [50] Xilinx, Inc., "All programmable SoC." Available: <http://www.xilinx.com/products/silicon-devices/soc/>
- [51] Xilinx, Inc., "Zynq UG479 7 series DSP user guide." Available: [http://www.xilinx.com/support/documentation/user\\_guides/](http://www.xilinx.com/support/documentation/user_guides/)
- [52] Xilinx, Inc., "Zynq UG585 technical reference manual." Available: [http://www.xilinx.com/support/documentation/user\\_guides/](http://www.xilinx.com/support/documentation/user_guides/)
- [53] J. Zhu and P. Sutton, "FPGA implementations of neural networks: A survey of a decade of progress," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2003, pp. 1062–1066.