SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration

Thierry Moreau Mark Wyse Jacob Nelson Adrian Sampson
University of Washington

Hadi Esmaeilzadeh
Georgia Institute of Technology

Luis Ceze Mark Oskin
University of Washington

***Abstract*—Many applications that can take advantage of accelerators are amenable to approximate execution. Past work has shown that neural acceleration is a viable way to accelerate approximate code. In light of the growing availability of on-chip field-programmable gate arrays (FPGAs), this paper explores neural acceleration on off-the-shelf programmable SoCs.**

We describe the design and implementation of SNNAP, a flexible FPGA-based neural accelerator for approximate programs. SNNAP is designed to work with a compiler workflow that configures the neural network's topology and weights instead of the programmable logic of the FPGA itself. This approach enables effective use of neural acceleration in commercially available devices and accelerates different applications without costly FPGA reconfigurations. No hardware expertise is required to accelerate software with SNNAP, so the effort required can be substantially lower than custom hardware design for an FPGA fabric and possibly even lower than current "C-to-gates" high-level synthesis (HLS) tools. Our measurements on a Xilinx Zynq FPGA show that SNNAP yields a geometric mean of 3.8× speedup (as high as 38.1×) and 2.8× energy savings (as high as 28×) with less than 10% quality loss across all applications but one. We also compare SNNAP with designs generated by commercial HLS tools and show that SNNAP has similar performance overall, with better resource-normalized throughput on 4 out of 7 benchmarks.

I. Introduction

In light of diminishing returns from technology improvements on performance and energy efficiency [20], [28], researchers are exploring new avenues in computer architecture. There are at least two clear trends emerging. One is the use of *specialized logic* in the form of accelerators [52], [53], [24], [27] or programmable logic [40], [39], [13], and another is *approximate computing*, which exploits applications' tolerance to quality degradations [44], [51], [21], [43]. Specialization leads to better efficiency by trading off flexibility for leaner logic and hardware resources, while approximate computing trades off accuracy to enable novel optimizations.

The confluence of these two trends leads to additional opportunities to improve efficiency. One example is *neural acceleration*, which trains neural networks to mimic regions of approximate code [22], [48]. 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) accelerator. This leads to better efficiency because neural networks are amenable to efficient hardware implementations [38], [19], [32], [45]. Prior work on neural acceleration, however, has assumed that the NPU is implemented in fully custom logic tightly integrated with the host processor pipeline [22], [48]. While modifying the CPU core to integrate the NPU yields significant performance and efficiency gains, it prevents near-term adoption and increases design cost/complexity. This paper explores the performance opportunity of NPU acceleration implemented on off-the-shelf *field-programmable gate arrays* (FPGAs) and without tight NPU–core integration, avoiding changes to the processor ISA and microarchitecture.

On-chip FPGAs have the potential to unlock order-of-magnitude energy efficiency gains while retaining some of the flexibility of general-purpose hardware [47]. Commercial parts that incorporate general purpose cores with programmable logic are beginning to appear [54], [2], [31]. In light of this trend, this paper explores an opportunity to accelerate approximate programs via an NPU implemented in programmable logic.

Our design, called SNNAP (**s**ystolic **n**eural **n**etwork **a**ccelerator in **p**rogrammable logic), is designed to work with a compiler workflow that automatically configures the neural network's topology and weights instead of the programmable logic itself. SNNAP's implementation on off-the-shelf programmable logic has several benefits. First, it enables effective use of neural acceleration in commercially available devices. Second, since NPUs can accelerate a wide range of computations, SNNAP can target many different applications without costly FPGA reconfigurations. Finally, the expertise required to use SNNAP can be substantially lower than designing custom FPGA configurations. In our evaluation, we find that the programmer effort can even be lower than for commercially available "C-to-gates" high-level synthesis tools [42], [18].

We implement and measure SNNAP on the Zynq [54], a state-of-the-art programmable system-on-a-chip (PSoC). We identify two core challenges: communication latency between the core and the programmable logic unit, and the difference in processing speeds between the programmable logic and the core. We address those challenges with a new throughput-oriented interface and programming model, and a parallel architecture based on scalable FPGA-optimized systolic arrays. To ground our comparison, we compare benchmarks accelerated with SNNAP to custom designs of the same accelerated code generated by a high-level synthesis tool. Our HLS study shows that current commercial tools still require significant effort and hardware design experience. Across a suite of approximate benchmarks, we observe an average speedup of 3.8×, ranging from 1.3× to 38.1×, and an average energy savings of 2.8×.

II. Programming

There are two basic ways to use SNNAP. The first is to use a high-level, compiler-assisted mechanism that transforms regions of approximate code to offload them to SNNAP. This automated

neural acceleration approach requires low programmer effort and is appropriate for bringing efficiency to existing code. The second is to directly use SNNAP's low-level, explicit interface that offers fine-grained control for expert programmers while still abstracting away hardware details. We describe both interfaces below.

A. Compiler-Assisted Neural Acceleration

Approximate applications can take advantage of SNNAP automatically using the *neural algorithmic transformation* [22]. This technique uses a compiler to replace error-tolerant subcomputations in a larger application with neural network invocations.

The process begins with an approximation-aware programming language in which code or data can be marked as approximable. Language options include Relax's code regions [17], EnerJ's type qualifiers [44], Rely's variable and operator annotations [9], or simple function annotations. In any case, the programmer's job is to express where approximation is allowed. The neural-acceleration compiler trains neural networks for the indicated regions of approximate code using test inputs. The compiler then replaces the original code with an invocation of the learned neural network. Lastly, quality can be monitored at run-time using application-specific quality metrics such as Light-Weight Checks [26].

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

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

where the programmer uses a function attribute to mark `filter()` as approximate.

The neural-acceleration compiler replaces the `filter()` call with instructions that instead invoke SNNAP with the argument `in_image[x][y]`. The compiler also adds setup code early in the program to set up the neural network for invocation.

B. Low-Level Interface

While automatic transformation represents the highest-level interface to SNNAP, it is built on a lower-level interface that acts both as a compiler target and as an API for expert programmers. 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 a processor 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. The 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 is most effective 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 ARMv7 `SEV` (send event) instruction to notify SNNAP. The accelerator then reads the inputs from the CPU's cache via a cache-coherent 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.

Low-Level asynchronous API. SNNAP's accompanying software library offers a low-level API that abstracts away the details of the hardware-level interface. The library provides an ordered, asynchronous API that hides the size of SNNAP's input and output buffers. This interface is useful both as a target for neural-acceleration compilers and for expert programmers who want convenient, low-level control over SNNAP.

The SNNAP C library uses a callback function to consume each output of the accelerator when it is ready. For example, a simple callback that writes a single floating-point output to an array can be written:

```
static int index = 0;
static float output[...];
void cbk(const void *data) {
  output[index] = *(float *)data; ++index;
}
```

Then, to invoke the accelerator, the program configures the library, sends inputs repeatedly, and then waits until all invocations are finished with a barrier. For example:

```
snnap_stream_t stream = snnap_stream_new(
  sizeof(float), sizeof(float), cbk);
for (int i = 0; i < max; ++i) {
  snnap_stream_put(stream, input);
}
snnap_stream_barrier(stream);
```

The `snnap_stream_new` call creates a stream configuration describing the size the neural network's invocation in bytes, the size of each corresponding output, and the callback function. Then, `snnap_stream_put` copies an input value from a **`void`**`*` pointer into SNNAP's memory-mapped input buffer. Inside the put call, the library also consumes any outputs available in SNNAP's output buffer and invokes the callback function if necessary. Finally, `snnap_stream_barrier` waits until all invocations are finished.

This asynchronous style enables the SNNAP runtime library to coalesce batches of inputs without exposing buffer management to the programmer or the compiler. The underlying SNNAP configuration can be customized with different buffer sizes without requiring changes to the code. In more sophisticated programs, this style also allows the program to transparently overlap SNNAP invocations with CPU code between `snnap_stream_send` calls.

This low-level, asynchronous interface is suitable for expert programmers who want to exert fine-grained control over how the program communicates with SNNAP. It is also appropriate for situations when the program explicitly uses a neural network model for a traditional purpose, such as image classification or handwriting recognition, where the SNNAP C library acts as a replacement for a software neural network library. In most

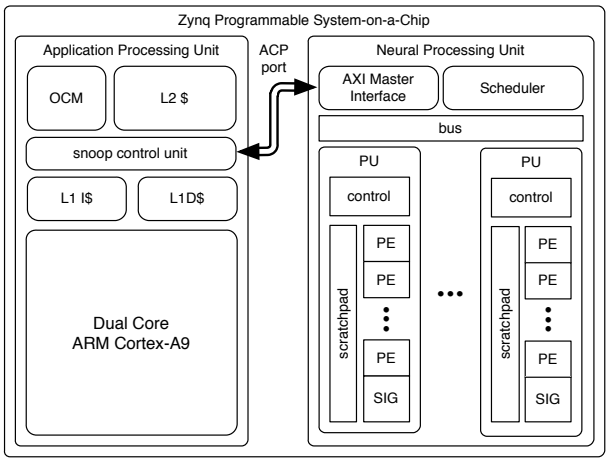


Fig. 1: SNNAP system diagram. Each Processing Unit (PU) contains a chain of Processing Elements (PE) feeding into a sigmoid unit (SIG).

cases, however, programmers need not directly interact with the library and can instead rely on automatic neural acceleration.

III. Architecture Design for SNNAP

This work is built upon an emerging class of heterogeneous computing devices called Programmable System-on-Chips (PSoCs). These devices combine a set of hard processor cores with programmable logic on the same die. Compared to conventional FPGAs, this integration provides a higher-bandwidth and lower-latency interface between the main CPU and the programmable logic. However, the latency is still higher than in previous proposals for neural acceleration [22], [48]. Our objective is to take advantage of the processor–logic integration with efficient invocations, latency mitigation, and low resource utilization. We focus on these challenges:

- The NPU must use FPGA resources efficiently to minimize its energy consumption.
- The NPU must support low-latency invocations to provide benefit to code with small approximate regions.
- To mitigate communication latency, the NPU must be able to efficiently process batches of invocations.
- The NPU and the processor must operate independently to enable the processor to hibernate and conserve energy while the accelerator is active.
- Different applications require different neural network topologies. Thus, the NPU must be reconfigurable to support a wide range of applications without the need for reprogramming the entire FPGA or redesigning the accelerator.

The rest of this section provides an overview of the SNNAP NPU and its interface with the processor.

A. SNNAP Design Overview

SNNAP evaluates *multi-layer perceptron* (MLP) neural networks. MLPs are a widely-used class of neural networks that have been used in previous work on neural acceleration [22], [48]. An MLP is a layered directed graph where the nodes are computational elements called *neurons*. Each neuron computes the weighted sum of its inputs and applies a nonlinear function, known as the *activation function*, to the sum—often a sigmoid function. The complexity of a neural network is reflected in its *topology:* larger topologies can fit more complex functions while smaller topologies are faster to evaluate.

The SNNAP design is based on *systolic arrays*. Systolic arrays excel at exploiting the regular data-parallelism found in neural networks [14] and are amenable to efficient implementation on modern FPGAs. Most of the systolic array's highly pipelined computational datapath can be contained within the dedicated multiply–add units found in FPGAs know as *Digital Signal Processing* (DSP) slices. We leverage these resources to realize an efficient pipelined systolic array for SNNAP in the programmable logic.

Our design, shown in Figure 1, consists of a cluster of *Processing Units* (PUs) connected through a bus. Each PU is composed of a control block, a chain of *Processing Elements* (PEs), and a sigmoid unit, denoted by the SIG block. The PEs form a one-dimensional systolic array that feeds into the sigmoid unit. When evaluating a layer of a neural network, PEs read the neuron weights from a local scratchpad memory where temporary results can also be stored. The sigmoid unit implements a nonlinear neuron-activation function using a lookup table. The PU control block contains a configurable sequencer that orchestrates communication between the PEs and the sigmoid unit. The PUs operate independently, so different PUs can be individually programmed to parallelize the invocations of a single neural network or to evaluate many different neural networks. Section IV details SNNAP's hardware design.

B. CPU–SNNAP Interface

We design the CPU–SNNAP interface to allow dynamic reconfiguration, minimize communication latency, and provide high-bandwidth coherent data transfers. To this end, we design a wrapper that composes three different interfaces on the target programmable SoC (PSoC).

We implement SNNAP on a commercially available PSoC: the Xilinx Zynq-7020 on the ZC702 evaluation platform [54]. The Zynq includes a Dual Core ARM Cortex-A9, an FPGA fabric, a DRAM controller, and a 256 KB scratchpad SRAM referred to as the on-chip memory (OCM). While PSoCs like the Zynq hold the promise of low-latency, high-bandwidth communication between the CPU and FPGA, the reality is more complicated. Zynq provides multiple communication mechanisms with different bandwidths and latencies that can surpass 100 CPU cycles. This latency can in some cases dominate the time it takes to evaluate a neural network. SNNAP's interface must therefore mitigate this communication cost with a modular design that permits throughput-oriented, asynchronous neural-network invocations while keeping latency as low as possible.

We compose a communication interface based on three available communication mechanisms on the Zynq PSoC [57]. First, when the program starts, it configures SNNAP using the medium-throughput General Purpose I/Os (GPIOs) interface. Then, to use SNNAP during execution, the program sends

inputs using the high-throughput ARM Accelerator Coherency Port (ACP). The processor then uses the ARMv7 `SEV`/`WFE` signaling instructions to invoke SNNAP and enter sleep mode. The accelerator writes outputs back to the processor's cache via the ACP interface and, when finished, signals the processor to wake up. We detail each of these components below.

Configuration via General Purpose I/Os (GPIOs). The ARM interconnect includes two 32-bit Advanced Extensible Interface (AXI) general-purpose bus interfaces to the programmable logic, 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 can only support moderate bandwidth. We use these GPIO interfaces to configure SNNAP after it is synthesized on the programmable logic. The program sends a configuration to SNNAP without reprogramming the FPGA. A configuration consists of a schedule derived from the neural network topology and a set of weights derived from prior neural network training. SNNAP exposes the configuration storage to the compiler as a set of memory-mapped registers. To configure SNNAP, the software checks that the accelerator is idle and writes the schedule, weights, and parameters to memory-mapped SRAM tables in the FPGA known as block RAMs.

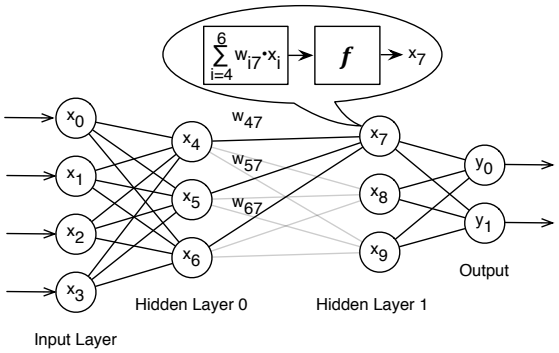
Sending data via the Accelerator Coherency Port. The FPGA can access the ARM on-chip memory system through the 64-bit Accelerator Coherency Port (ACP) AXI-slave interface. This port allows the FPGA to send read and write requests directly to the processors' Snoop Control Unit to access the processor caches thus bypassing explicit cache flushes required by traditional DMA interfaces. The ACP interface is the best available option for transferring batches of input/output vectors to and from SNNAP. SNNAP includes a custom AXI master for the ACP interface, reducing round-trip communication latency down to 93 CPU cycles. Batching invocations help amortize this latency in practice.

Invocation via synchronization instructions. The ARM and the FPGA are connected by two unidirectional event lines `eventi` and `evento` for synchronization. The ARMv7 ISA contains two instructions to access these synchronization signals, `SEV` and `WFE`. The `SEV` instruction causes the `evento` signal in the FPGA fabric to toggle. The `WFE` instruction causes the processor to enter the low-power hibernation state until the FPGA toggles the `eventi` signal. These operations have significantly lower latency (5 CPU cycles) than any of the other two communication mechanisms between the processor and the programmable logic.

We use these instructions to invoke SNNAP and synchronize its execution with the processor. To invoke SNNAP, the CPU writes input vectors to a buffer in its cache. It signals the accelerator to start computation using `SEV` and enters hibernation with `WFE`. When SNNAP finishes writing outputs to the cache, it signals the processor to wake up and continues execution.

IV. Hardware Design for SNNAP

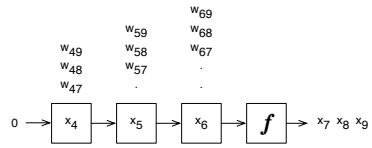
This section describes SNNAP's systolic-array design and its FPGA implementation.



(a) An multilayer perceptron neural network.

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

(b) Matrix representation of hidden layer evaluation.



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

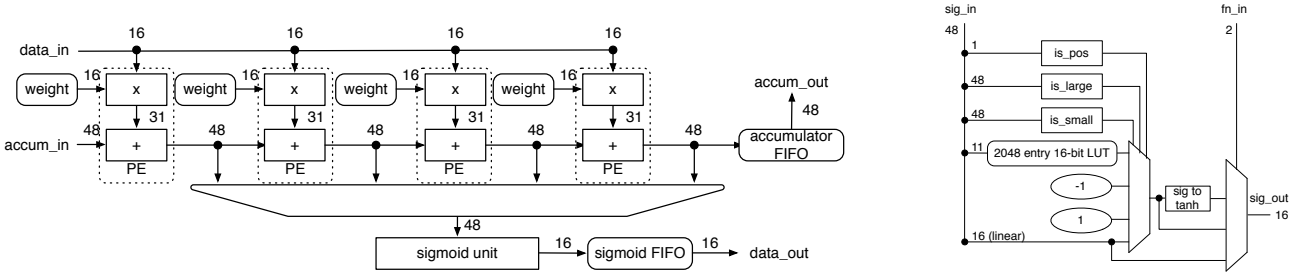
Fig. 3: Implementing multi-layer perceptron neural networks with systolic arrays.

A. Multi-Layer Perceptrons With Systolic Arrays

MLPs consist of a collection of neurons organized into layers. Figure 3a depicts an MLP with four layers: the input layer, the output layer, and two *hidden layers*. The computation of one of the neurons in the second hidden layer is highlighted: the neuron computes the weighted sum of the values of its source neurons and applies the activation function f to the result. The resulting neuron output is then sent to the next layer.

The evaluation of an MLP neural network consists of 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–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 while the matrix elements elements w_{ji} trickle in. Each cell performs a multiplication $x_i \cdot w_{ji}$, adds it to the sum of products produced by the upstream cell to its left, and sends the result to the downstream cell to its right. The output vector produced by the systolic array finally goes through an activation function cell, completing the layer computation.

Systolic arrays can be efficiently implemented using the hard DSP slices that are common in modern FPGAs. Our PSoC incorporates 220 DSP slices in its programmable logic [57]. DSP slices offer pipelined fixed-point multiply-and-add functionality and a hard-wired data bus for fast aggregation



(a) Processing Unit datapath.

(b) Sigmoid Unit datapath.

Fig. 2: Detailed PU datapath: PEs are implemented on multiply–add logic and produce a stream of weighted sums from an input stream. The sums are sent to a sigmoid unit that approximates the activation function.

of partial sums on a single column of DSP slices. As a result, a one-dimensional fixed-point systolic array can be contained entirely in a single hard logic unit to provide high performance at low power [56].

B. Processing Unit Datapath

Processing Units (PUs) are replicated processing cores in SNNAP's design. A PU comprises a chain of *Processing Elements* (PEs), a sigmoid unit, and local memories including block-RAMs (BRAMs) and FIFOs that store weights and temporary results. A sequencer orchestrates communication between the PEs, the sigmoid unit, local memories, and the bus that connects each PU to the NPU's memory interface.

The PEs that compose PUs map directly to a systolic array cell as in Figure 2a. A PE consists of a multiply-and-add module implemented on a DSP slice. The inputs to the neural network are loaded every cycle via the input bus into each PE following the systolic algorithm. Weights, on the other hand, are statically partitioned among the PEs in local BRAMs.

The architecture can support an arbitrary number of PEs. Our evaluation discusses the optimal number of PEs per PU by discussing throughput-resources trade-offs.

Sigmoid unit. The sigmoid unit applies the neural network's activation function to outputs from the PE chain. The design, depicted in Figure 2b, is a 3-stage pipeline comprising a lookup-table and some logic for special cases. We use a $y = x$ linear approximation for small input values and $y = \pm 1$ for very large inputs. Combined with a 2048-entry LUT, the design yields at most 0.01% normalized RMSE.

SNNAP supports three commonly-used activation functions: a sigmoid function $S(x) = \frac{k}{1+e^{-x}}$, a hyperbolic tangent $S(x) = k \cdot \tanh(x)$, and a linear activation function $S(x) = k \cdot x$, where k is a steepness parameter. Microcode instructions (see Section IV-C) dictate the activation function for each layer.

Flexible NN topology. The NPU must map an arbitrary number of neurons to a fixed number of PEs. Consider a layer with n input neurons, m output neurons and let p be the number of PEs in a PU. Without any constraints, we would schedule the layer on n PEs, each of which would perform m multiplications. However, p does not equal n in general. When $n < p$, there are excess resources and $p - n$ PEs remain idle. If $n > p$, we time-multiplex the computation onto the p PEs by storing temporary sums in an *accumulator FIFO*. Section IV-C details the process of mapping layers onto PEs.

A similar time-multiplexing process is performed to evaluate neural networks with many hidden layers. We buffer sigmoid unit outputs in a *sigmoid FIFO* until the evaluation of the current layer is complete; then they can be used as inputs to the next layer. When evaluating the final layer in a neural network, the outputs coming from the sigmoid unit are sent directly to the memory interface and written to the CPU's memory.

The BRAM space allocated to the sigmoid and accumulator FIFOs limit the maximum layer width of the neural networks that SNNAP can execute.

Numeric representation. SNNAP uses a 16-bit signed fixed-point numeric representation with 7 fraction bits internally. This representation fits within the 18×25 DSP slice multiplier blocks. The DSP slices also include a 48-bit fixed-point adder that helps avoid overflows on long summation chains. We limit the dynamic range of neuron weights during training to match this representation.

The 16-bit width also makes efficient use of the ARM core's byte-oriented memory interface for applications that can provide fixed-point inputs directly. For floating-point applications, SNNAP converts the representation at its inputs and outputs.

C. Processing Unit Control

Microcode. SNNAP executes a static schedule derived from the topology of a neural network. This inexpensive scheduling process is performed on the host machine before it configures the accelerator. The schedule is represented as microcode stored in a local BRAM.

Each microcode line describes a command to be executed by a PE. We distinguish architectural PEs from physical PEs since there are typically more inputs to each layer in a neural network than there are physical PEs in a PU (i.e., $n > p$). Decoupling the architectural PEs from physical PEs allow us to support larger neural networks and makes the same micro-code executable on PUs of different PE length.

Each instruction comprises four fields:

Schedule	FU	0	1	2	3	4	5	6	7
Naive	PE_0	$x_2^{(0)}$	$x_3^{(0)}$			$x_4^{(0)}$		$x_2^{(1)}$	$x_3^{(1)}$
	PE_1		$x_2^{(0)}$	$x_3^{(0)}$			$x_4^{(0)}$		$x_2^{(1)}$
	SIG			$x_2^{(0)}$	$x_3^{(0)}$			$x_4^{(0)}$	
Efficient	PE_0	$x_2^{(0)}$	$x_3^{(0)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(0)}$	$x_4^{(1)}$		$x_2^{(2)}$
	PE_1		$x_2^{(0)}$	$x_3^{(0)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(0)}$	$x_4^{(1)}$	
	SIG			$x_2^{(0)}$	$x_3^{(0)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(0)}$	$x_4^{(1)}$

TABLE I: Static PU scheduling of a 2–2–1 neural network. The naive schedule introduces pipeline stalls due to data dependencies. Evaluating two neural network invocations simultaneously by interlacing the layer evaluations can eliminate those stalls.

1) `ID`: the ID of the architectural PE executing the command.
2) `MADD`: the number of multiply–add operations that must execute to compute a layer.
3) `SRC`: input source selector; either the input FIFO or the sigmoid FIFO.
4) `DST`: the destination of the output data; either the next PE or the sigmoid unit. In the latter case, the field also encodes (1) the type of activation function used for that layer, and (2) whether the layer is the output layer.

Sequencer. The sequencer is a finite-state machine that processes microcoded instructions to orchestrate data movement between PEs, input and output queues, and the sigmoid unit within each PU. Each instruction is translated by the sequencer into commands that get forwarded to a physical PE along with the corresponding input data. The mapping from architectural PE (as described by the microcode instruction) to the physical PE (the actual hardware resource) is done by the sequencer dynamically based on resource availability and locality.

Scheduler optimizations. During microcode generation, we use a simple optimization that improves utilization by minimizing pipeline stalls due to data dependencies. The technique improves overall throughput for a series of invocations at the cost of increasing the latency of a single invocation.

Consider a simple PU structure with two PEs and a one-stage sigmoid unit when evaluating a 2–2–1 neural network topology. Table I presents two schedules that map this neural network topology onto the available resources in the pipeline diagram. Each schedule tells us which task each functional unit is working on at any point in time. For instance, when PE_1 is working on x_2, it is multiplying $x_1 \times w_{12}$ and adding it to the partial sum $x_0 \times w_{02}$ computed by PE_0.

Executing one neural network invocation at a time results in a inefficient schedule as illustrated by the *naive schedule* in Table I. The pipeline stalls here result from (1) dependencies between neural network layers and (2) contention over the PU input bus. Data dependencies occur when a PE is ready to compute the next layer of a neural network, but has to wait for the sigmoid unit to produce the inputs to that next layer.

We eliminate these stalls by interleaving the computation of layers from multiple neural network invocations as shown in the *efficient schedule* in Table I. Pipeline stalls due to data dependencies can be eliminated as long as there are enough neural network invocations waiting to be executed. SNNAP's throughput-oriented workloads tend to provide enough invocations to justify this optimization.

V. Evaluation

We implemented SNNAP on an off-the-shelf programmable SoC. In this section, we evaluate our implementation to assess its performance and energy benefits over software execution, to characterize the design's behavior, and to compare against a high-level synthesis (HLS) tool. The HLS comparison provides a reference point for SNNAP's performance, efficiency, and programmer effort requirements.

A. Experimental setup

Applications. Table II shows the applications measured in this evaluation, which are the benchmarks used by Esmaeilzadeh et al. [22] along with `blackscholes` from the PARSEC benchmark suite [6]. We offload one approximate region from each application to SNNAP. These regions are mapped to neural network topologies used in previous work [22], [11]. The table shows 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 III. The ZYNQ processor integrates a mobile-grade ARM Cortex-A9 and a Xilinx FPGA fabric on a single TSMC 28nm die.

We compiled our benchmarks using GCC 4.7.2 at its `-O3` optimization level. We ran the benchmarks directly on the bare metal processor.

Monitoring performance and power. To count CPU cycles, we use the event counters in the ARM's architectural performance monitoring unit and 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 on each of the board's power planes. This allows us to track power usage for the different sub-systems (e.g., CPU, FPGA, DRAM).

NPU configuration. Our results reflect a SNNAP configuration with 8 PUs, each comprised of 8 PEs. The design runs at 167 MHz, or 1/4 of the CPU's 666MHz frequency. For each benchmark, we configure all the PUs to execute the same neural network workload.

High-Level Synthesis infrastructure. We use Vivado HLS 2014.2 to generate hardware kernels for each benchmark. We then integrate the kernels into SNNAP's bus interface and program the FPGA using Vivado Design Suite 2014.2.

B. Performance and Energy

This section describes the performance and energy benefits of using SNNAP to accelerate our benchmarks.

Performance. Figure 4a shows the whole application speedup when SNNAP is used to execute each benchmark's target region, while the rest of the application runs on the CPU, over an all-CPU baseline.

Application	Description	Error Metric	NN Topology	NN Config. Size	Error	Amdahl Speedup (×)
`blackscholes`	option pricing	mean error	6–20–1	6308 bits	7.83%	> 100
`fft`	radix-2 Cooley-Tukey FFT	mean error	1–4–4–2	1615b	0.1%	3.92
`inversek2j`	inverse kinematics for 2-joint arm	mean error	2–8–2	882b	1.32%	> 100
`jmeint`	triangle intersection detection	miss rate	18–32–8–2	15608b	20.47%	99.65
`jpeg`	lossy image compression	image diff	64–16–4	21264b	1.93%	2.23
`kmeans`	*k*-means clustering	image diff	6–8–4–1	3860b	2.55%	1.47
`sobel`	edge detection	image diff	9–8–1	3818b	8.57%	15.65

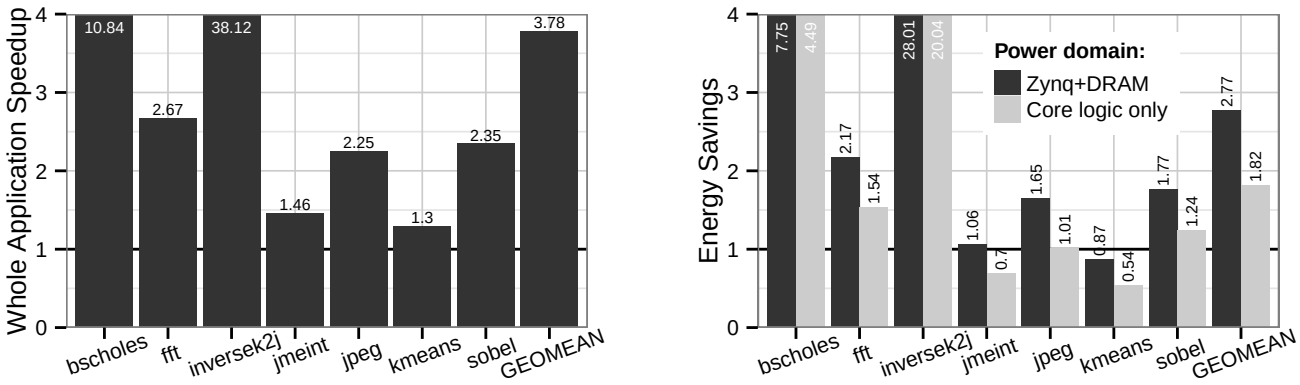
TABLE II: Applications used in our evaluation. The "NN Topology" column shows the number of neurons in each MLP layer. The "NN Config. Size" column reflects the size of the synaptic weights and microcode in bits. "Amdahl Speedup" is the hypothetical speedup for a system where the SNNAP invocation is instantaneous.

Zynq SoC	
Technology	28nm TSMC
Processing	2-core Cortex-A9
FPGA	Artix-7
FPGA Capacity	53KLUTs, 106K Flip-Flops
Peak Frequencies	667MHz A9, 167MHz FPGA
DRAM	1GB DDR3-533MHz

Cortex-A9	
L1 Cache Size	32kB I$, 32kB D$
L2 Cache Size	512kB
Scratch-Pad	256kB SRAM
Interface Port	AXI 64-bit ACP
Interface Latency	93 cycles roundtrip

NPU	
Number of PUs	8
Number of PEs	8
Weight Memory	1024×16-bit
Sigmoid LUT	2048×16-bit
Accumulator FIFO	1024×48-bit
Sigmoid FIFO	1024×16-bit
DSP Unit	16×16-bit multiply, 48-bit add

TABLE III: Microarchitectural parameters for the Zynq platform, CPU, FPGA and NPU.



(a) Speedup

(b) Energy savings

Fig. 4: Performance and energy benefit of SNNAP acceleration over an all-CPU baseline execution of each benchmark.

The average speedup is 3.78×. Among the benchmarks, `inversek2j` has the highest speedup (38.12×) since the bulk of the application is offloaded to SNNAP, and the target region of code includes trigonometric function calls that take over 1000 cycles to execute on the CPU and that a small neural network can approximate. Conversely, `kmeans` sees only a 1.30× speedup, mostly because the target region is small and runs efficiently on a CPU, while the corresponding neural network is relatively deep.

Energy. Figure 4b shows the energy savings for each benchmark over the same all-CPU baseline. We show the savings for two different energy measurements: (1) the SoC with its DRAM and other peripherals, and (2) the core logic of the SoC. On average, neural acceleration with SNNAP provides a 2.77× energy savings for the SoC and DRAM and a 1.82× savings for the core logic alone.

The *Zynq+DRAM* evaluation shows the power benefit from using SNNAP on a chip that already has an FPGA fabric. Both measurements include all the power supplies for the Zynq chip and its associated DRAM and peripherals, including the FPGA. The FPGA is left unconfigured for the baseline.

The *core logic* evaluation provides a conservative estimate of the potential benefit to a mobile SoC designer who is considering including an FPGA fabric in her design. We compare a baseline consisting only of the CPU with the power of the CPU and FPGA combined. No DRAM or peripherals are included.

On all power domains and for all benchmarks except `jmeint` and `kmeans`, neural acceleration on SNNAP results in energy savings. In general, the more components we include in our power measurements, the lower the relative power cost and the higher the energy savings from neural acceleration. `inversek2j`, the benchmark with the highest speedup, also has the highest energy savings. For `jmeint` and `kmeans` we observe a decrease in energy efficiency in the core logic measurement; for `kmeans`, we also see a decrease in the Zynq+DRAM measurement. While the CPU saves power by sleeping while SNNAP executes, the accelerator incurs more

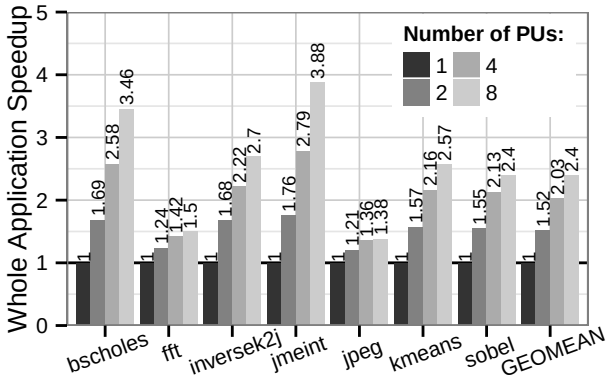


Fig. 5: Performance of neural acceleration as the number of PUs increase.

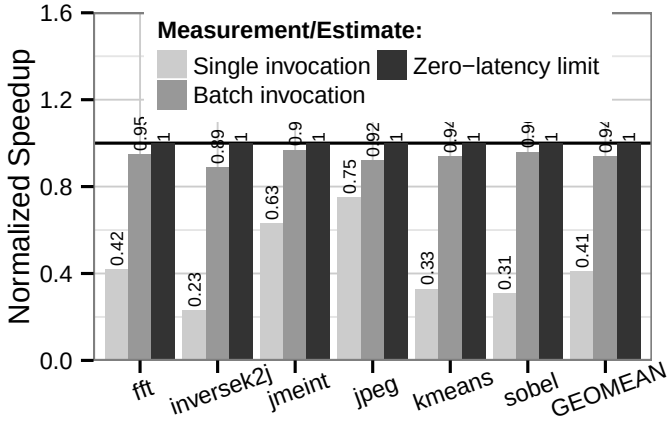


Fig. 6: Impact of batching on speedup.

power than this saves, so a large speedup is necessary to yield energy savings.

C. Characterization

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

Impact of parallelism. Figure 5 shows the performance impact of SNNAP's parallel design by varying the number of PUs. On average, increasing from 1 PU to 2 PUs, 4 PUs, and 8 PUs improves performance by 1.52×, 2.03×, and 2.40× respectively. The `sobel`, `kmeans` and `jmeint` benchmarks require at least 2, 4, and 8 PUs respectively to see any speedup.

Higher PU counts lead to higher power consumption, but the cost can be offset by the performance gain. The best energy efficiency occurs at 8 PUs for most benchmarks. The exceptions are `jpeg` and `fft`, where the best energy savings are with 4 PUs. These benchmarks have a relatively low "Amdahl speedup limit" , so they see diminishing returns from parallelism.

Impact of batching. Figure 6 compares the performance of batched SNNAP invocations, single invocations, and zero-latency invocations - an estimate of the speedup if there were no communication latency between the CPU and the accelerator.

With two exceptions, non-batched invocations lead to a slowdown due to communication latency. Only `inversek2j` and `jpeg` see a speedup since their target regions are large enough to outweigh the communication latency. Comparing with the zero-latency estimate, we find that batch invocations are effective at hiding this latency. Our 32-invocation batch size is within 11% of the zero-latency ideal.

Optimal PE count. Our primary SNNAP configuration uses 8 PEs per PU. A larger PE count can decrease invocation latency but can also have lower utilization, so there is a trade-off between fewer, larger PUs or more, smaller PUs given the same overall budget of PEs. In Figure 7a, we examine this trade-off space by sweeping configurations with a fixed number of PEs. The NPU configurations range from 1 PU consisting of 16 PEs (1×16) through 16 PUs each consisting of a single PE (16×1). The 16×1 arrangement offers the best throughput. However, resource utilization is not constant: each PU has control logic and memory overhead. The 16×1 NPU uses more than half of the FPGA's LUT resources, whereas the 2×8 NPU uses less than 4% of all FPGA resources. Normalizing throughput by resource usage (Figure 7b) indicates that the 2×8 configuration is optimal.

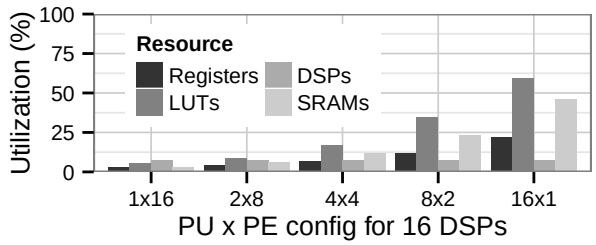
D. Design Statistics

FPGA utilization. Figure 7c shows the FPGA fabric's resource utilization for varying PU counts. A single PU uses less than 4% of the FPGA resources. The most utilized resources are the slice LUTs at 3.92% utilization and the DSP units at 3.64%. With 2, 4, 8, and 16 PUs, the design uses less than 8%, 15% 30% and 59% of the FPGA resources respectively and the limiting resource is the DSP slices. The approximately linear scaling reflects SNNAP's balanced design.

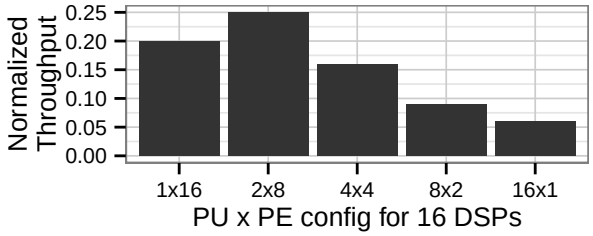
Memory Bandwidth. Although the Zynq FPGA can accommodate 16 PUs, the current ACP interface design does not satisfy the bandwidth requirements imposed by compute-resource scaling for benchmarks with high bandwidth requirements (e.g. `jpeg`). This limitation is imposed by the ACP port used to access the CPUs cache hierarchy. During early design exploration, we considered accessing memory via higher-throughput non-coherent memory ports, but concluded experimentally that at a fine offload granularity, the frequent cache flushes were hurting performance. As a result, we evaluate SNNAP at 8-PUs to avoid being memory bound by the ACP port. We leave interface optimizations and data compression schemes that could increase effective memory bandwidth as future work.

Output quality. We measure SNNAP's effect on output quality using application-specific error metrics, as is standard in the approximate computing literature [44], [21], [22], [46]. Table II lists the error metrics.

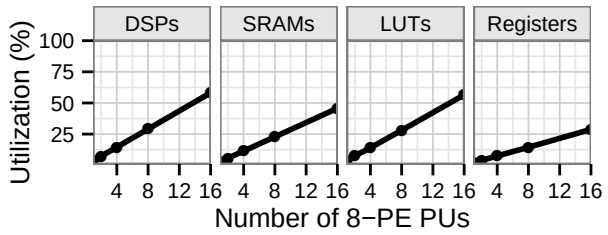
We observe less than 10% application output error for all benchmarks except `jmeint`. `jmeint` had high error due to complicated control flow within the acceleration region, but we include this benchmark to fairly demonstrate the applicability of neural acceleration. Among the remaining applications, the highest output error occurs in `sobel` with 8.57% mean absolute pixel error with respect to a precise execution.



(a) Static resource utilization for multiple configurations of 16 DSP units.



(b) Peak throughput on jmeint normalized to most-limited FPGA resource for each configuration.



(c) Static resource utilization of multiple 8-PE PUs.

Fig. 7: Exploration of SNNAP static resource utilization.

Application	Effort	Clock	Pipelined	Util.
`blackscholes`	3 days	148 MHz	yes	37%
`fft`	2 days	166 MHz	yes	10%
`inversek2j`	15 days	148 MHz	yes	32%
`jmeint`	5 days	66 MHz	no	39%
`jpeg`	5 days	133 MHz	no	21%
`kmeans`	2 days	166 MHz	yes	3%
`sobel`	3 days	148 MHz	yes	5%

TABLE IV: HLS-kernel specifics per benchmark: required engineering time (working days) to accelerate each benchmark in hardware using HLS, kernel clock, whether the design was pipelined, most-utilized FPGA resource utilization.

E. HLS Comparison Study

We compare neural acceleration with SNNAP against Vivado HLS [55]. For each benchmark, we attempt to compile using Vivado HLS the same target regions used in neural acceleration. We synthesize a precise specialized hardware datapath and integrate it with the same CPU–FPGA interface we developed for SNNAP and contrast whole-application speedup, resource-normalized throughput, FPGA utilization, and programmer effort.

Speedup.

Table IV shows statistics for each kernel we synthesized with Vivado HLS. The kernels close timing between 66 MHz and 167 MHz (SNNAP runs at 167 MHz). We compare the performance of the HLS-generated hardware kernels against SNNAP.

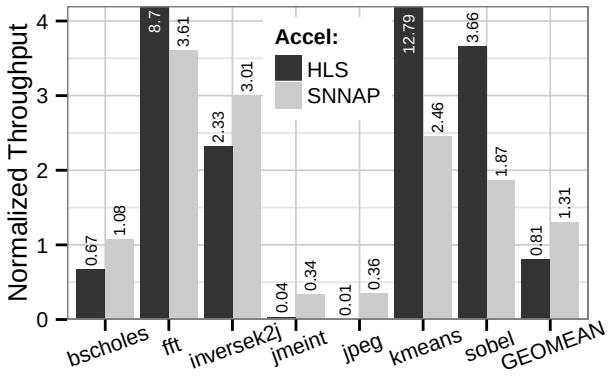


Fig. 9: Resource-normalized throughput of the NPU and HLS accelerators.

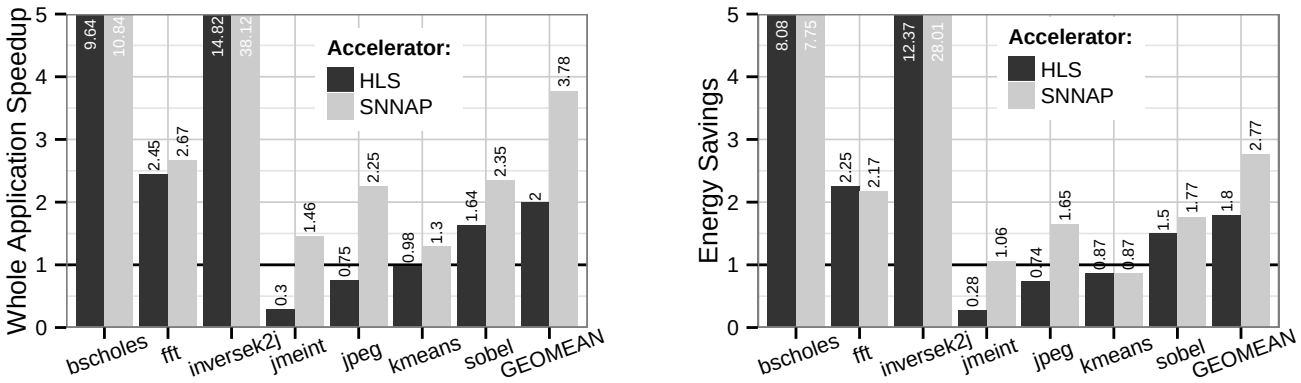
Figure 8a shows the whole-application speedup for HLS and SNNAP. The NPU outperforms HLS on all benchmarks, yielding a 3.78× average speedup compared to 2× for HLS. The `jmeint` benchmark provides an example of a kernel that is not a good candidate for HLS tools; its dense control flow leads to highly variable evaluation latency in hardware, and the HLS tool was unable to pipeline the design. Similarly, `jpeg` performs poorly using HLS due to DSP resource limitations on the FPGA. Again, the HLS tool was unable to pipeline the design, resulting in a kernel with long evaluation latency.

Resource-normalized kernel throughput. To assess the area efficiency of SNNAP and HLS, we isolate FPGA execution from the rest of the application. We compute the theoretical throughput (evaluations per second) by combining the *pipeline initiation interval* (cycles per evaluation) from functional simulation and the f_{max} (cycles/second) from post-place-and-route timing analysis. We obtain post-place-and-route resource utilization by identifying the most-used resource in each design. The resource-normalized throughput is the ratio of these two metrics.

Figure 9 compares the resource-normalized throughput for SNNAP and HLS-generated hardware kernels. Neural acceleration does better than HLS for `blackscholes`, `inversek2j`, `jmeint` and `jpeg`. In particular, while HLS provides better absolute throughput for `blackscholes` and `inversek2j`, the kernels also use an order of magnitude more resources than a single SNNAP PU. `kmeans` and `sobel` have efficient HLS implementations with utilization roughly equal to one SNNAP PU, resulting in 2–5× greater throughput.

Programming experience. "C-to-gates" tools are promoted for their ability to hide the complexity of hardware design. With our benchmarks, however, we found hardware expertise to be essential for getting good results using HLS tools. Every benchmark required hardware experience to verify the correctness of the resulting design and extensive C-code tuning to meet the tool's requirements.

Table IV lists the number of working days required for a student to produce running hardware for each benchmark using



(a) Single HLS kernel and 8-PU NPU whole-application speedups over CPU-only execution baseline.

(b) Energy savings of single HLS kernel and 8-PU NPU over CPU-only baseline for Zynq+DRAM power domain.

Fig. 8: Performance and energy comparisons of HLS and SNNAP acceleration.

HLS. The student is a Masters researcher with Verilog and hardware design background but not prior HLS experience. Two months of work was needed for familiarization with the HLS tool and the design of a kernel wrapper to interact with SNNAP's custom memory interface. After this initial cost, compiling each benchmark took between 2 and 15 days. `blackscholes`, `fft`, `kmeans`, and `sobel` all consist of relatively simple code, and each took only a few days to generate fast kernels running on hardware. The majority of the effort was spent tweaking HLS compiler directives to improve pipeline efficiency and resource utilization. Accelerating `jmeint` was more involved and required 5 days of effort, largely spent attempting (unsuccessfully) to pipeline the design. `jpeg` also took 5 days to compile, which was primarily spent rewriting the kernel's C code to make it amenable to HLS by eliminating globals, precomputing lookup tables, and manually unrolling some loops. Finally, `inversek2j` required 15 days of effort. The benchmark used the arc-sine and arc-cosine trigonometric functions, which are not supported by the HLS tools, and required rewriting the benchmark using mathematical identities with the supported arc-tangent function. The latter exposed a bug in the HLS workflow that was eventually resolved by upgrading to a newer version of the Vivado tools.

Discussion. While HLS offers a route to FPGA use without approximation, it is far from flawless: significant programmer effort and hardware-design expertise is still often required. In contrast, SNNAP acceleration uses a single FPGA configuration and requires no hardware knowledge. Unlike HLS approaches, which place restrictions on the kind of C code that can be synthesized, neural acceleration treats the code as a black box: the internal complexity of the legacy software implementation is irrelevant. SNNAP's FPGA reconfiguration-free approach also avoids the overhead of programming the underlying FPGA fabric, instead using a small amount of configuration data that can be quickly loaded in to accelerate different applications. These advantages make neural acceleration with SNNAP a viable alternative to traditional C-to-gates approaches.

VI. 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 [4], [46], [3], [35], [36], [30] or hardware [17], [34], [21], [33], [37], [10], [22], [44], [26] techniques. SNNAP represents the first work (to our knowledge) to exploit this trade-off using tightly integrated 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 [22], [48] or direct manual configuration [11], [50], [5]. This work has typically assumed special-purpose neural-network hardware; SNNAP represents an opportunity to realize these benefits on commercially available hardware. Recent work has proposed combining neural transformation with GPU acceleration to unlock order-of-magnitude speedups by elimiating control flow divergence in SIMD applications [25], [26]. This direction holds a lot of promise in applications where a large amount of parallelism is available. Until GPUs become more tightly integrated with the processor core, their applicability remains limited in applications where the invocation latency is critical (i.e. small code offload regions). Additionally the power envelope of GPUs has been traditionally high. Our work targets low power accelerators and offers higher applicability by offloading computation at a finer granularity than GPUs.

Hardware support for neural networks. There is an extensive body of work on hardware implementation of neural networks both in digital [38], [19], [58], [12], [16], [7] and analog [8], [45], [49], [32] domains. Other work has examined fault-tolerant hardware neural networks [29], [50]. There is also significant prior effort on FPGA implementations of neural networks ([58] contains a comprehensive survey). Our contribution is a design

that enables automatic acceleration of approximate software without engaging programmers in hardware design.

FPGAs as accelerators. This work also relates to work on synthesizing designs for reconfigurable computing fabrics to accelerate traditional imperative code [40], [41], [15], [23]. Our work leverages FPGAs by mapping diverse code regions to neural networks via neural transformation and accelerating those code regions onto a fixed hardware design. By using neural networks as a layer of abstraction, we avoid the complexities of hardware synthesis and the overheads of FPGA compilation and reconfiguration. Existing commercial compilers provide means to accelerate general purpose programs [55], [1] with FPGAs but can require varying degrees of hardware expertise. Our work presents a programmer-friendly alternative to using traditional "C-to-gates" high-level synthesis tools by exploiting applications' tolerance to approximation.

VII. Conclusion

SNNAP enables the use of programmable logic to accelerate approximate programs without requiring hardware design. Its high-throughput systolic neural network mimics the execution of existing imperative code. We implemented SNNAP on the Zynq system-on-chip, a commercially available part that pairs CPU cores with programmable logic and demonstrate 3.8× speedup and 2.8× energy savings on average over software execution. The design demonstrates that approximate computing techniques can enable effective use of programmable logic for general-purpose acceleration while avoiding custom logic design, complex high-level synthesis, or frequent FPGA reconfiguration.

VIII. Acknowledgments

The authors thank the anonymous reviewers for their throrough comments on improving this work. The authors also thank the SAMPA group for their useful feedback, and specifically Ben Ransford, Andre Baixo for adopting SNNAP in their to-be-published compiler support for approximate accelerators work. The authors thank Eric Chung for his help on prototyping accelerators on the Zynq. This work was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA, SRC contract #2014-EP-2577, the Qualcomm Innovation Fellowship, NSF grant #1216611, the NSERC, and gifts from Microsoft Research and Google.

References

[1] Altera Corporation, "Altera OpenCL Compiler." Available: http://www.altera.com/products/software/opencl/

[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] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008, pp. 451–460.

[7] H. T. Blair, J. Cong, and D. Wu, "Fpga simulation engine for customized construction of neural microcircuits," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2013.

[8] B. E. Boser, E. Säckinger, J. Bromley, Y. Lecun, L. D. Jackel, and S. Member, "An analog neural network processor with programmable topology," *Journal of Solid-State Circuits*, vol. 26, no. 12, pp. 2017–2025, December 1991.

[9] 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.

[10] 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 *Design, Automation and Test in Europe (DATE)*, 2006, pp. 1110–1115.

[11] 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 *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, Nov 2012, pp. 36–45.

[12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[13] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big data on little clients," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*, 2013, pp. 261–272.

[14] 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: Elsevier Science Publishers B. V., May 1992, pp. 145–159. Available: http://dx.doi.org/10.1016/0165-6074(92)90017-2

[15] 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.

[16] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, "Deep learning with COTS HPC systems," 2013.

[17] M. de Kruijf, 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.

[18] G. de Micheli, Ed., *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] B. Grigorian and G. Reinman, "Accelerating divergent applications on simd architectures using neural networks," in *International Conference on Computer Design (ICCD)*, 2014.

[26] B. Grigorian and G. Reinman, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *Conference on Adaptive Hardware and Systems (AHS)*, 2014.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Intel Corporation, "Disrupting the data center to create the digital services economy." Available: https://communities.intel.com/community/itpeernetwork/datastack/blog/2014/06/18/disrupting-the-data-center-to-create-the-digital-services-economy

[32] 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.

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

[34] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: 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.

[35] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," MIT, Tech. Rep. MIT-CSAIL-TR-2010-038, Aug. 2010.

[36] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *Static Analysis Symposium (SAS)*, 2011.

[37] 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.

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

[39] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014, pp. 13–24.

[40] 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.

[41] 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.

[42] S. Safari, A. H. Jahangir, and H. Esmaeilzadeh, "A parameterized graph-based framework for high-level test synthesis," *Integration, the VLSI Journal*, vol. 39, no. 4, pp. 363–381, Jul. 2006.

[43] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[44] 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.

[45] 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.

[46] 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.

[47] S. Sirowy and A. Forin, "Where's the beef? why FPGAs are so fast," Microsoft Research, Tech. Rep. MSR-TR-2008-130, Sep. 2008.

[48] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 505–516. Available: http://dl.acm.org/citation.cfm?id=2665671.2665746

[49] 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.

[50] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 356–367.

[51] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.

[52] 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.

[53] 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.

[54] Xilinx, Inc., "All programmable SoC." Available: http://www.xilinx.com/products/silicon-devices/soc/

[55] Xilinx, Inc., "Vivado high-level synthesis." Available: http://www.xilinx.com/products/design-tools/vivado/

[56] Xilinx, Inc., "Zynq UG479 7 series DSP user guide." Available: http://www.xilinx.com/support/documentation/user_guides/

[57] Xilinx, Inc., "Zynq UG585 technical reference manual." Available: http://www.xilinx.com/support/documentation/user_guides/

[58] 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.