# Compilation and Hardware Support for Approximate Acceleration

Thierry Moreau Adrian Sampson Andre Baixo Mark Wyse Ben Ransford Jacob Nelson Luis Ceze Mark Oskin
University of Washington

***Abstract*—Approximate computing exposes opportunities to increase the efficiency of computer systems by trading-off accuracy for energy savings. These optimization opportunities can be found in many emerging classes of applications which exhibit a degree of tolerance to imprecision. Taking full advantage of approximation requires a programming model that provides control to the programmer over what portions of a program can be approximated, as well as hardware-based techniques that can effectively trade-off accuracy for energy savings. We describe ACCEPT, a compiler framework for approximate programs and SNNAP, an approximate accelerator that can efficiently offload regions of approximate code. Using ACCEPT, a software programmer can leverage the benefits of approximate computing by annotating software with data-centric annotations and letting the compiler identify the best approximation strategies based on accuracy requirements. We evaluate programs that offload approximate computation on SNNAP, and observe an average speedup of 3.8× and an average energy saving of 2.8×.**

## I. Introduction

Energy consumption is a first-class concern in computer system designs. In warehouse-scale computers, reducing power demands can drastically impact operating costs. In mobile systems, battery technology is advancing slowly. Making mobile devices more capable from generation to generation will necessitate new ways to extract more from each joule of a battery's capacity.

*Approximate computing* is an emerging research area that promises to offer drastic energy savings. Approximate computing exploits the fact that many applications do not require perfect correctness. Many important mobile applications use "soft" error-tolerant computations, including computer vision, sensor data analysis, machine learning, augmented reality, signal processing, and search. A few small errors while detecting faces or displaying game graphics, for example, may be acceptable or even unnoticeable, yet today's systems faithfully compute precise outputs even when the inputs are imprecise.

Approximate computing research builds software and hardware that are allowed to make mistakes when applications are willing to tolerate them. Approximate systems can reclaim energy that is currently lost to the "correctness tax" imposed by traditional safety margins designed to prevent worst-case scenarios.

In order to safely eliminate this "correctness tax", programmers need a tractable way to write correct software even when the hardware can be incorrect. For example, and image renderer can tolerate errors in the pixel data it outputs; a small number of erroneous pixels may be acceptable or even undetectable. However, an error in a jump table could lead to a crash, and even small errors in the image file format might make the output unreadable. By letting the programmer isolate parts of the program that must be precise from those that can be approximated, the compiler can ensure that the program functions correctly even as quality degrades.

In addition to programming models that provide safety in approximate programs, eliminating the "correctness tax" requires mechanisms that can smoothly trade-off accuracy for performance. These mechanisms must provide substantial performance gains to justify the annotation effort required to make a program safe to approximate. For instance, an approximate adder might not provide large enough of an energy saving due to the large control and data movement overheads typically found in general purpose processors, and therefore would not constitute a compelling candidate for an approximation strategy.

In an effort to make it practical and compelling for software programmers to leverage quality-energy trade-offs, we co-designed a programming model, a compiler framework, and a hardware co-processor to support approximate computing. Our end-to-end system includes two distinct building blocks:

- A new programmer-guided compiler framework transforms programs to use approximation in a controlled way. ACCEPT, an Approximate C Compiler for Energy and Performance Trade-offs, uses programmer annotations, static analysis, and dynamic profiling to find parts of a program that are amenable to approximation.
- An approximate accelerator prototype that can efficiently evaluate coarse regions of approximate code. SNNAP, a Systolic Neural Network Accelerator in Programmable logic, is a hardware accelerator prototype that can efficiently evaluate approximate regions of code in a general-purpose program. The prototype is implemented on the FPGA (Field Programmable Gate Array) fabric of an off-the-shelf ARM SoC, which makes its near-term adoption possible. Hardware acceleration with SNNAP is enabled by *neural acceleration* [2], an algorithmic transformation that substitutes regions of code in a program with approximate versions amenable to efficient evaluation on specialized hardware.

Using ACCEPT and SNNAP, a software programmer can leverage the benefits of approximate computing in a disciplined manner. Evaluating SNNAP over 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. An Approximate Compiler

ACCEPT[1] is compiler framework for approximate computing [5]. It combines programmer annotations, code analysis, optimizations, and profiling feedback to make approximation safe and keep control in the hands of programmers.

ACCEPT's frontend, build atop the LLVM compiler infrastructure, extends the syntax of C and C++ to incorporate an `APPROX` keyword that programmers use to annotate data types. ACCEPT's analysis identifies code that can affect only variables marked as `APPROX`. Optimizations use these analysis results to avoid transforming the precise parts of the program. An autotuning component measures program executions and uses heuristics to identify program variants that maximize performance and output quality. The final output is a set of Pareto-optimal versions of the input program that reflect its efficiency–quality trade-off space.

*Safety constraints and feedback.* Because program relaxations can have outsize effects on program behavior, programmers need *visibility* into—and *control* over—the transformations the compiler applies. To give the programmer fine-grained control over relaxations, ACCEPT extends an existing lightweight annotation system for approximate computing based on type qualifiers [4]. ACCEPT gives programmers visibility into the relaxation process via feedback that identifies which transformations can be applied and which annotations are constraining it. Through annotation and feedback, the programmer iterates toward an annotation set that unlocks new performance benefits while relying on an assurance that critical computations are unaffected.

*Automatic program transformations.* Based on programmer annotations, ACCEPT's compiler passes apply program transformations that involve only approximate data. To this end, ACCEPT provides a compiler analysis library that finds regions of code that are amenable to transformations. An ensemble of optimization strategies transform these regions. One critical optimization targets SNNAP, our neural accelerator, which we describe in more detail below.

*Autotuning.* While a set of annotations may permit many different safe program relaxations, not all of them are beneficial in the quality–performance trade-off they offer. A practical approximation mechanism must help programmers choose from among many candidate relaxations for a given program to strike an optimal balance between performance and quality. ACCEPT's autotuner heuristically explores the space of possible relaxed programs to identify Pareto-optimal variants.

## III. Neural Acceleration

A powerful approach to approximate computing, *neural acceleration*, works by substituting entire regions of code in a program with machine learning models [2]. Neural acceleration trains neural networks to mimic and replace regions of approximate imperative 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) accelerator. Neural networks have efficient hardware implementations, so this workflow can offer significant energy savings over traditional execution.

[1]Source available at http://accept.rocks

Neural acceleration consists of three phases: programming, compilation, and execution.

*Programming.* To use neural acceleration in ACCEPT, the programmer uses profiling information and type annotations to mark data that is amenable to approximation. For many applications, it is easy to identify the "core" approximate data, such as the pixel array in an image filter algorithm, that dominates the program's execution. The programmer also provides a *quality metric* that measures the accuracy of the program's overall output.

*Compilation.* The compiler implements neural acceleration in four phases: region selection, execution observation, training, and code generation. ACCEPT first identifies large regions of code that are safe to approximate and nominates them as candidates for neural acceleration. Next, it executes the program with test cases and records the inputs and outputs to each target code region. It then uses this input–output data to train a neural network that mimics the original code. Training can use standard techniques for neural networks: we use the standard *backpropagation* algorithm. Finally, the compiler generates an executable that replaces the original code with invocations of a special accelerator, the neural processing unit (NPU), that implements the trained neural network.

*Execution.* During deployment, the transformed program begins execution on the main core and configures the NPU. Throughout execution, the program invokes the NPU to perform a neural network evaluation in lieu of executing the code region it replaced. Invoking the NPU is faster and more energy-efficient than executing the original code region on the CPU, so the program as a whole runs faster.

## IV. SNNAP: Hardware Support for Approximate Acceleration

Our NPU implementation, SNNAP [3], runs on off-the-shelf *field-programmable gate arrays* (FPGAs). Using existing, affordable hardware means that SNNAP can provide benefit today, without waiting for new silicon. SNNAP uses 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 with special-purpose hardware [2]. Our design covers this additional latency with a batching mechanism. In addition, we compensate the mismatch between CPU frequency and FPGA frequency with added parallelism in the accelerator design.

*Implementation on the Zynq.* We have implemented SNNAP on a commercially available PSoC: the Xilinx Zynq-7020 on the ZC702 evaluation platform [6]. The Zynq includes a Dual Core ARM Cortex-A9 and an FPGA fabric. The CPU–NPU interface composes three communication mechanisms on the Zynq PSoC [7] for high bandwidth and low latency. 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

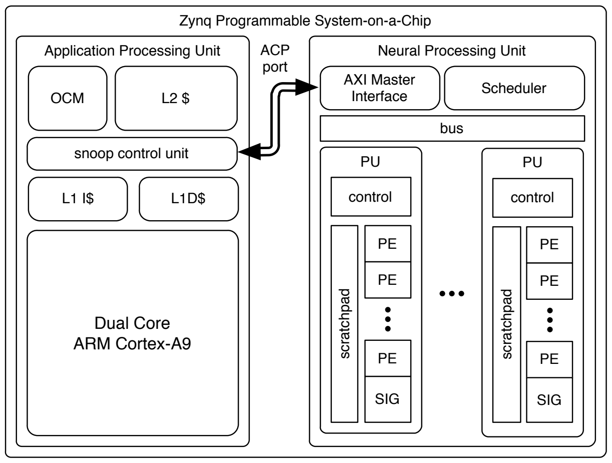

Fig. 1: SNNAP system diagram. Each Processing Unit (PU) contains a chain of Processing Elements (PE) feeding into a sigmoid unit (SIG).

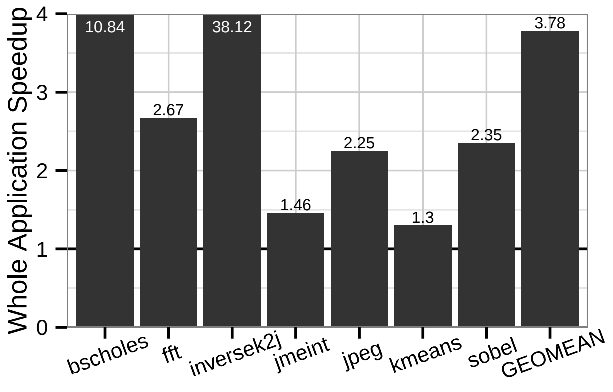

Fig. 2: Speedup

Fig. 3: Performance benefit of neural acceleration with SNNAP over an all-CPU baseline execution of each benchmark.

instructions to invoke SNNAP and enter sleep mode. Finally, the accelerator writes outputs back to the processor's cache via the ACP interface and, when finished, signals the processor to wake up.

*Micro-Architecture.* 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. Systolic arrays excel at exploiting the regular data-parallelism found in neural networks, and are amenable to efficient implementation on modern FPGAs. 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 can be programmed to operate independently, so different PUs can either be used to parallelize the invocations of a single neural network or used to evaluate different neural networks concurrently.

## V. Experience and Results

We have applied ACCEPT and SNNAP to a set of approximable benchmarks. Our goal was to show that programmers can unlock significant efficiency gains at a tolerable accuracy cost with low programming effort.

### A. Writing Approximate Programs

To evaluate the effort required to apply approximation, we annotated a set of benchmarks for ACCEPT's language. The programmers included three undergraduate researchers, all of whom were beginners with the C and C++ languages and new to approximate computing, as well as graduate students more familiar with the field.

Programmers tended to approach annotation by finding the central approximable data in the program—e.g., the vector coordinates in a clustering algorithm, or pixels in imaging code. ACCEPT's type errors guided programmers toward other parts of the code that needed annotation. Programmers needed to balance effort with potential reward during annotation, so auxiliary tools like profilers and call graph generators were useful to find hot spots.

Overall, ACCEPT was able to detect candidate regions of code in the majority of benchmarks solely based on data annotations. For these benchmarks, ACCEPT was capable of instrumenting code automatically to take advantage of neural acceleration. For the remaining benchmarks, the programmers have to use an internal API to make use of neural acceleration.

### B. SNNAP Acceleration Efficiency

Our evaluation targeted 7 benchmarks from many application domains: option pricing (`blackscholes`), signal processing (`fft`), robotics (`inversek2j`), lossy image compression (`jpeg`), machine learning (`kmeans`) and image processing (`sobel`). Table I provides insight on the size of the neural network that had to be trained to approximate the computation in each benchmarks.

We compared each program's performance, power, and energy consumption when using SNNAP versus running the original software on the ARM processor. We limited each application's output error to 10%. SNNAP incorporates 8 processing units and runs at one quarter of the CPU's core frequency.

*Performance and Energy Efficiency.* Figure 2 shows geometric mean whole-application speedup of 3.78× across our benchmark suite ranging from 1.30× for `kmeans` to 38.12× for `inversek2j`. Inverse kinematics saw the largest speedup since the bulk of its execution was offloaded to SNNAP. The target code for that benchmark includes trigonometric function calls that are expensive to evaluate on an ARM CPU. Neural acceleration approximates these functions with a compact neural network that can be quickly evaluated on SNNAP. Conversely, `kmeans` had the lowest speedup because a small fraction of the program execution got offloaded to SNNAP. Also, the neural network that approximates target region in `kmeans` was relatively deep and did not present a significant advantage over executing the precise code on the CPU.

| 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 I: 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.

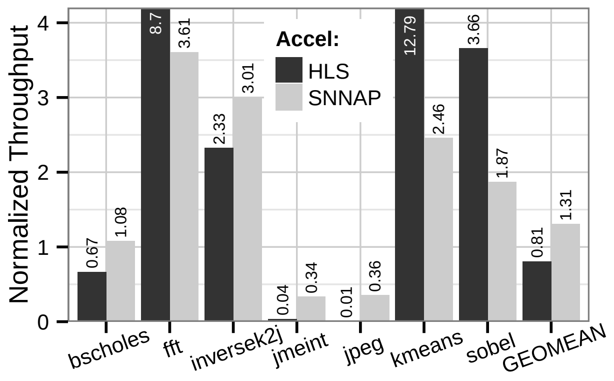

Fig. 4: Resource-normalized throughput of the NPU and HLS accelerators.

The energy efficiency results were similar: 2.77× energy reduction on the SoC+DRAM subsystem ranging from 0.87× for `kmeans` to 28.01× for `inversek2j`. The primary energy benefit of SNNAP comes from racing to completion: SNNAP has a fixed power overhead which has to be compensated for by faster execution times.

*Output quality.* We measure SNNAP's effect on output quality using application-specific error metrics, as is standard in the approximate computing literature [4], [1], [2]. Table I 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.

*Comparing SNNAP with Fixed-Function Accelerators.* Specialized hardware accelerators are another way to improve applications' energy efficiency using FPGAs. We compared SNNAP's performance to specialized FPGA designs generated by a commercial high-level synthesis tool, Vivado HLS 2014.2. For each benchmark, we generate a specialized datapath by compiling through HLS the same region of code that we offload to SNNAP via neural acceleration.

For a fair comparison, we normalize the performance of each approach by its resource usage on the FPGA. The results of this study are shown in Figure 4. To our surprise, neural acceleration offers better resource-normalized throughput on 4 out of 7 benchmarks. These 4 benchmarks generally has larger datapaths with sometimes high control flow divergence, which made it difficult for HLS tools to fit a fully pipelined datapath on the available FPGA resources.

Aside from competitive performance, SNNAP and ACCEPT also offer *programmability* and *generality* advantages over specialized datapaths. Neural acceleration does not require hardware design knowledge which was often required to debug or optimize the performance of HLS kernels. Also, all benchmarks we compiled through HLS generated a different datapath, whereas SNNAP provides a single fabric for accelerating all 7 benchmarks, making virtualization possible.

## VI. Future Work

ACCEPT and SNNAP represent the first steps toward near-term approximate computing on PSoCs. But compilation and neural acceleration are not the only challenges in approximate computing. We are also developing high-level tools to help programmers better navigate and understand performance–quality trade-offs, including special-purpose debuggers for approximate programs. We also wish to explore the rich design space for approximate acceleration; neural acceleration is just one coarse-grained technique among others. Future work will also establish better error guarantees for neural acceleration.

## VII. Conclusion

Many important classes of applications can tolerate some imprecision. Programs from diverse domains such as machine learning, vision and embedded sensing exhibit trade-offs between accuracy and energy efficiency. Approximate program transformations such as neural acceleration have been shown to drastically improve performance and energy usage while only minimally impacting output quality. By providing tools to reason about quality relaxation, we can ensure that approximate transformations do not have destructive effects on program execution.

ACCEPT is a compiler framework for approximate programming that balances automation with programmer insight. ACCEPT makes approximate accelerators useful by helping programmers reason about quality–performance trade-offs. SNNAP, our approximate accelerator prototype on an off-the-shelf ARM SoC with programmable logic, demonstrates a 3.8× speedup and 2.8× energy savings for approximate applications.

Approximate computing research is in its infancy and needs more tools for prototyping and evaluating ideas. The ACCEPT framework and the SNNAP prototype demonstrate a practical and efficient implementation of approximate transformation.

## REFERENCES

[1] 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.

[2] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” *Micro, IEEE*, vol. 33, no. 3, pp. 16–27, 2013.

[3] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, “SNNAP: Approximate computing on programmable socs via neural acceleration,” in *HPCA*, Feb. 2015.

[4] 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.

[5] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, “ACCEPT: A programmer-guided compiler framework for practical approximate computing,” Univesity of Washington, Tech. Rep. UW-CSE-15-01-01, Jan. 2015.

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

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