1. **Approximate Computing Techniques For FPGAs**

One of the techniques for FPGAs is presented by Moreau et al. [1]. They present a technique for neural acceleration of approximable codes on a programmable system on chip (SoC), that is, an off-the-shelf field-programmable gate array (FPGA). Their approach explores the performance opportunity of neural processing unit (NPU) acceleration implemented on off-the-shelf FPGAs and without tight NPU–core integration, avoiding changes to the processor ISA and micro-architecture.

With the help of technology improvement on performance and energy efficiency, researchers are exploring new avenues in computer architecture. Two of the emerging trends are specialized logic in the form of accelerators or programmable logic, and approximate computing, which exploits applications’ tolerance to quality degradations. 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. Once the neural network is trained, the system no longer executes the original code and instead invokes the neural network model on a NPU accelerator. This leads to better efficiency because neural networks are amenable to efficient hardware implementations. However, prior work on neural acceleration has assumed that the NPU is implemented in fully custom logic tightly integrated with the host processor pipeline. 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 technique faces these challenges.

There are two basic ways to use their technique (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. Approximate applications can take advantage of SNNAP automatically using the neural algorithmic transformation. This technique uses a compiler to replace error-tolerant sub-computations 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. 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. 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. 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.[1]

Another technique is presented by Sampson and his fellows [2]. Their approach is ACCEPT (an Approximate C Compiler for Energy and Performance Trade-offs), a framework for approximation that balances automation with programmer guidance. ACCEPT automatically applies a variety of approximation techniques, including hardware acceleration, while ensuring their safety. They apply ACCEPT to nine workloads on a standard desktop, an FPGA-augmented mobile SoC, and an energy-harvesting sensor device to evaluate the annotation process.

Recent work has shown how to accelerate approximate programs with hardware neural networks. Neural acceleration uses profiled inputs and outputs from a region of code to train a neural network that mimics the code. The original code is then replaced with an invocation of an efficient hardware accelerator implementation, the NPU. But the technique has thus far required manual identification of candidate code regions and insertion of offloading instructions. ACCEPT automates the process.

ACCEPT safely and efficiently harnesses the potential of approximate programs by combining three main techniques: (1) a programmer–compiler feedback loop consisting of source code annotations and an analysis log; (2) a compiler analysis library that enables a range of automatic program relaxations; and (3) an autotuning system that uses dynamic measurements of candidate program relaxations to find the best balances between efficiency and quality. The final output is a set of Pareto-optimal versions of the input program that reflect its efficiency–quality trade-off space. ACCEPT implements an automatic neural acceleration transform that uses an existing configurable neural-network implementation for an on-chip FPGA , which is based on the previous technique. ACCEPT uses approximate region selection (x4.2) to identify acceleration targets, then trains a neural network on execution logs for each region. It then generates code to offload executions of the identified region to the accelerator. The offload code hides invocation latency by constructing batched invocations that exploit the high-bandwidth interface between the CPU and FPGA. This techniques targets a commercially available FPGA-augmented SoC and does not require specialized neural hardware.[2]

[1] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. *SNNAP: Approximate computing on programmable SoCs via neural acceleration*. In International Symposium on High Performance Computer Architecture (HPCA’15). 603–614.

[2] Adrian Sampson, Andr´e Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. *ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing*. Technical Report UW-CSE-15-01-01. University of Washington, Seattle, WA.