Off-the-shelf Tensor Processing Unit as Approximate Computing Accelerator

Christopher Russell
Escuela Electrica y Mecanica
Universidad Latina de Costa Rica
Heredia, Costa Rica
christopher.russell@ulatina.net

Pamela Salazar
Escuela de Ingenieria Electronica
Technological Institude of Costa Rica
Cartago, Costa Rica
p.salazar@estudiantec.cr

Abstract*—We examine the suitability of an off-the-shelf Tensor Processing Unit as an accelerator for approximate computing, with the hope that it may replace custom approximate hardware or complicated FPGA implementations. For this, we choose the EdgeTPU found on the Google Coral Dev Board, which accelerates Tensor Flow Lite models. Our efforts were somewhat complicated by the immature nature of the libraries and tools for the use of Tensor Flow Lite with C/C++ applications. Initially we approximate the standard AxBench CPU test cases using the Parrot transformation on the identified approximable regions with no algorithm modification. Initial results are poor with slow execution, possibly due to a need for batching of model inferences to improve throughput, and we are unable to make a definitive evaluation of the differences in power and energy needs between the exact CPU-based and approximate TPU-based implementations.

I. Introduction

A. Approximate Computing

Modern computing applications involving pattern recognition, data mining, and synthesis depend on very large and increasing amounts of data, which in turn demands very large and increasing amounts of energy and resources to process that data.[24] This is at odds with the simultaneous demand for increased energy efficiency in the computing industry.[19]

For many of these demanding applications an optimal "exact" result is not required, or even possible. In most cases suboptimal "approximate" results within a degree of error are sufficient. These applications may be classified as [11]:

1) Applications with noisy real-world inputs
2) Applications with analog outputs for human interpretation
3) Applications without unique answers
4) Large Iterative/convergent applications

These applications either already include a degree of error or uncertainty, or are tolerant of the introduction of small amounts of error or uncertainty in their processing. For such applications, the emerging field of approximate computing offers tools to provide meaningful results while saving resources, such as energy, space, or time, as compared to an exact computation.

The lower level of accuracy required by the application compared to that available from the computing system represents an opportunity for increased efficiency with less exact computation methods by exploiting this accuracy gap. The intention is that the resource saving still represents an overall gain in efficiency beyond the impact of the increased level of error due to approximation.

The approximate techniques used may be based in hardware where less accurate but more efficient circuitry is used, or in software where non-critical operations or memory accesses are eliminated.[11]

There exist implementation techniques of approximate computing at the circuit level, the architectural level, and the algorithmic level.[14]

At the architectural level, circuit-level techniques are used in the design of dedicated approximate accelerators, or to implement approximation-aware instruction sets in general purpose accelerators. However, due to their complexity approximate hardware solutions are usually described and analyzed in isolation, rather than in complete architectures. Few architecture-level solutions are described in the literature or appear to have been implemented in silicon.[4]

An interesting algorithmic technique is to model discrete sections of code as neural networks and train them with observed inputs during operation. Following the training of the network, the original exact code implementation is replaced by the trained neural network, which necessarily produces approximate results.[3]

Sanyal et al.[18] largely list hardware options for approximation in deep neural network implementation, however they also note the potential of algorithmic techniques such as pruning and reduced-complexity networks to lead to reduced hardware design.

Making such techniques even more attractive is the emergence of general purpose neural network hardware accelerators in edge applications, such as the Google Edge TPU. These accelerators implement neural networks in specialized parallel hardware, reducing time and power consumption compared to CPU-based implementations.[1][20]

Significant opportunities exist for investigation into the use of emerging low-power machine learning hardware accelerators as approximate computing devices. A combination of these may allow the possibility of implementation of approximation by algorithmic and hardware techniques through automatic means.

As an alternative to bespoke hardware design, programmable logic using FPGAs has been suggested recently

for the implementation of neural networks in MCU-FPGA SoC devices or using FPGA-based accelerators. The implementation overhead associated with the use of such devices is high, however.[20]

Software-based acceleration techniques operate at a very fine-grained level, requiring significant algorithm knowledge and programmer effort. There is opportunity for the development and application of minimally-invasive coarse-grained techniques which do not require low-level developer intervention, and allow for the high-level replacement of functional blocks.

The field of deep learning offers a potential solution, through the use of the Parrot transformation and similar techniques. Combined with the emergence of ASIC-based deep learning accelerators and their compatible APIs, such techniques could also use dedicated hardware greatly improve their performance using off-the-shelf components.

II. Literature Review

A. Measurement of Approximation

a) AxBench: Yazdanbakhsh et al offer Axbench[25] as a suite of reference applications suitable for comparative evaluation of approximation systems. Providing a diverse set of representative applications from a range of applicable domains, along with different input data sets and application-specific quality metrics for each, they claim that AxBench facilitates benchmarking and comparison of approximation techniques in the covered domains. Recognizing that different techniques will have different impacts on different workloads, AxBench provides common benchmarks, as well as benchmarks specific to CPUs, GPUs, and hardware approximation designs.

AxBench may be used to evaluate the quality impact of a technique by comparing the approximate and exact results using the provided quality metric for each benchmark, as well as to evaluate the gains of the technique in a particular implementation.

B. Accelerable Approximation Techniques

a) Parrot Transformation: ASICs, GPUs and FPGAs are among the alternatives for hardware acceleration of applications. ASICs are highly efficient, however their high cost and long development times makes them rigid and unsuitable for emerging applications. GPUs allow massive SIMD parallelism but perform poorly when threads diverge in their operations. FPGAs offer fine-grained and irregular parallelism, but their performance suffers when memory access is required. The Parrot transformation [3] seeks to avoid these limitations by using off-the shelf neural network accelerators to accelerate general-purpose approximate computing operations in applications which do not traditionally employ deep learning.

The workflow of the Parrot transformation is illustrated in Fig 1. During programming, sections of approximable code are identified and instrumented. During the compilation phase, this code is stimulated with training data, and the outputs of the approximable code regions are used to train a neural network. Once trained, this structure of this network is implemented

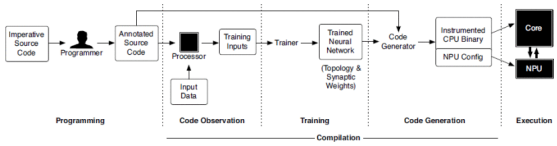


Fig. 1: Parrot transformation workflow. [25]

in a hardware accelerator, and the original approximable executable code is replaced by an interface to the accelerator. Input data is passed to the accelerator and the neural network estimates the approximate response.

For the Parrot transformation to be effective, the approximable region of code should be frequently executed, however there is little restriction on the size of the approximable region. Any body of code may be implemented in the accelerator regardless of its internal control flow, as long as it has a fixed number of inputs and outputs, and its processing does not produce changes in internal state or side-effects. The approximated function should not be chaotic, in that its output behavior should be well described by a reasonably sized set of training inputs.[25]

b) SNNAP: In their design SNNAP (Systolic Neural Network Accelerator in Programmable Logic), Moreau et al. [16] describe a compiler workflow that automatically produces the topology of an accelerating neural network, rather than a programmable logic design. This abstracts away the hardware implementation details and allows for the use of many off-the-shelf commercial devices.

At a high level, design begins with the programmer annotating approximable code using a suitable language. The compiler then trains networks for the indicated regions of code, as with the Parrot transformation, and replaces the original approximable code with a function call to the accelerator API. Design may also begin at a lower level using the accelerator API directly.

The authors note that loosely-integrated accelerators such as those based in an FPGA cannot block execution while computing individual outputs, due to their latency. The SNNAP project was developed for Zync SoCs including an on-die FPGA, which has higher banwidth and lower latency than a conventional off-chip FPGA, yet still is much slower than a tightly-integrated neural accelerator. High latency particularly affects code with small approximate regions. To improve throughput, SNNAP batches neural network accesses and sends them in bulk. At the API level, SNNAP implements an asynchronous mechanism using callbacks which hides the underlying hardware buffer mechanism from the programmer. This allows the SNNAP implementation to use varying buffer sizes without changes to the code, or to overlap SNNAP calls with CPU code. The accelerator design is able to support a wide range of applications and neural topologies without requiring FPGA reprogramming.

c) Off-the-shelf Neural Accelerator: González-Aragón and Castro-Godínez[5] describe an experiment with the use of the Parrot transformation to accelerate applications us-

ing a Raspberry Pi 4[15] with a Movidius Myriad X[12] neural accelerator on an Intel Neural Compute Stick 2[13]. Although they show promising improvements to be possible for a compute-intensive scene classification application, they note that the performance increase was insignificant for less intensive applications due to the low I/O bandwidth of the USB-connected accelerator.

III. Methodology

We propose to evaluate the suitability of the Google Coral Edge TPU for its use as an approximate computing accelerator using the Parrot transform. According to Google, the Edge TPU is a power-efficient inference accelerator ASIC for use in edge applications[7]. It is available with a PCIe interfaces, avoiding the USB bottleneck noted by González-Aragón and Castro-Godínez[5].

We choose to evaluate the Edge TPU using the Coral Dev Board, which features a quad-core Cortex-A53 CPU with an Edge TPU coprocessor[6]. On this platform we should be able to compare the energy performance of algorithms calculated exactly on the CPU, with their approximate implementations calculated on the TPU.

Our methodology is to boostrap the Coral and a development environment, implement approximate forms of appropriate cases from the AxBench test suite, and compare performance on the Coral between their exact and approximate forms.

A. Environment Provisioning

The provisioning of the environment consists of bootstrapping the Coral Dev Board, provisioning a development server for cross-compilation, and the cross-compilation of the Tensor Flow Lite and LibEdgeTPU libraries suitable for implementing models on the EdgeTPU devices.

1) Bootstrapping Coral Dev Board: Bootstrapping the Coral Dev Board is described in the "Getting Started" documentation[9]. Additionally we elected to format and mount an additional filesystem on a 128 GB microSD card for additional local storage. On this mount a 1 GB swap file was also created to supplement the 1 GB of physical RAM available on the board. Additionally a remote NFS filesystem hosted by the development server was mounted to facilitate easier copying of data between the server and target.

2) Development Server: As the Coral itself has limited resources and therefore is very slow to compile, development was done on a separate server and cross-compiled for the target platform. For this we used a VirtualBox VM running Ubuntu Linux. To simplify the setup of the cross-compilation environment, we elected to install an older version of Ubuntu (Ubuntu 10) contemporaneous with Mendel. Since both operating systems use the same version of glibc, executables cross-compiled on the development server with the aarch64-linux-gnu suite were then compatible with the glibc dynamic libraries in Mendel on the target. This eliminates the need to build a complete sandboxed development environment and only requires installation of the cross compilation tools from the "crossbuild-essential-arm64" Ubuntu package.

3) TensorFlow Lite Library/C++ API: Despite being intended for use with TensorFlow Lite networks, the Mendel OS for the Coral does not come with TensorFlow lite installed, and users are expected to install the version they require. TensorFlow Lite is primarily used through its Python interface, but for this project we are integrating with C++ applications, and so we require the TensorFlow Lite C++ API as well. The C++ API is not availble in binary form, and must be built by the user.

The TensorFlow build instructions[21] recommend using the Bazel build system, developed by Google, which purports to follow the hermetic principle ensuring that external dependencies are strictly controlled and builds are guaranteed to be repeatable. In practice, we found errors due to incompatibilities between versions of Bazel, and issues in our cross compilation environment due to unspecified dependencies on system include files. After many attempts to resolve the issue, we abandoned the use of Bazel and elected to use the alternative CMake instructions[22]. A CMake toolchain file for cross-compilation was adapted from the work of github user vpetrigo[23], and used to successfully build the `libtensorflow-lite.a` static library and the C/C++ API `libtensorflowlite_c.so` shared library.

The shared library was copied to `/usr/local/lib/libtensorflowlite_c.so` on the Coral.

4) LibEdgeTPU: The libedgetpu library provides the required interface between TensorFlow Lite and the Edge TPU hardware. Mendel comes installed with an obsolete version of the shared library, however we discovered that for use with the TensorFlow Lite C++ API, a new version must be compiled to be binary compatible with the particular version in use. Following the instructions in the project `README.md`[2], we cross-compiled `libedgetpu.so.1` in a Docker container. In order to do this, some tweaking of the `Dockerfile` was required, as the Debian version of the container is now obsolete, and the URLs of its APT repositories have changed to the archive server.

The shared library was copied to `/usr/lib/aarch64-linux-gnu/libedgetpu.so.1.0` on the Coral, replacing the version included in the distribution.

IV. Model Development

TensorFlow Lite models implementing the approximate versions of the AxBench functions were developed following the Parrot transformation methodology.

A. AxBench Observation

For the observation phase, the AxBench CPU Applications cases[10] were downloaded and run in a desktop environment, after converting their scripts for compatibility with Python 3. During the observation phase, each application produced a `aggregated.fann` file containing a list of inputs and corresponding outputs in ASCII format of the approximable code segment.

These observations supply the data for the training of the approximate models of the Parrot transformation.

B. Approximate Model Training

The observed data for each application was divided into training sets (90% of samples) and verification sets (10%) of samples, and used to train a TensorFlow Keras model in a Python script. The topology for each network was selected to match that used by the authors of the original Parrot transformation paper[3]. No particular care was dedicated to choosing an optimal topology or optimizing results at this stage, as the quantization of the model is expected to cause results to vary significantly, and research into the impact of topology on energy performance/accuracy and associated tuning techniques will be performed as part of a later work. The figure 2 shows the networks performance.

Following the training of the Keras model, the script calls `TFLiteConverter.from_keras_model` to convert to a TensorFlow Lite model. During this operation, we perform post-training quantization to convert the model to a wholly 8-bit integer form suitable for execution on the Edge TPU which only supports 8-bit integers. The input and output data types are not touched and remain as floats.

The EdgeTPU Compiler[8] is then used to convert the TensorFlow Lite model into a format suitable for loading into the EdgeTPU. All the TensorFlow Lite operations performed within the simple models produced are converted to TPU operations, except for the input and output quantization steps. Since these remain as floats, the quantization operation is performed within the model by the CPU, and no application code modification is required.

C. Application Modification

For convenience, we used the approximated source-code generated by the AxBench tests as the starting point for the TPU accelerated implementation. The AxBench scripts use the Fast Artificial Neural Network Library (FANN) library[17] to approximate the approximable locations identified by compiler pragmas. We manual replace the calls with our own code to execute our model.

We provide three wrapper functions for the TensorFlow Lite/EdgeTPU function calls:

```
void tfInit(const char* filename);
void tfClose(void);
void tfInfer(int numin, float* inPtr,
  int numout, float* outPtr);
```

- `tfInit()` accepts the filename of an EdgeTPU model, and initializes the EdgeTPU hardware, binds it to the interpreter, and loads the model.
- `tfClose()` unloads the interpreter and gracefully closes the EdgeTPU hardware
- `tfInfer()` accepts an input vector, executes it on the model, and returns the resulting output vector in the location indicated.

Calls to `tfInit()` and `tfClose()` are placed at the beginning and end of the `main()` function of the application, while the FANN calls in the approximable region are replaced by a call to `tfInfer()`. In this initial attempt we are only transferring a single vector in and out on each inference, rather than batching vectors of multiple instances, as this would require algorithm changes. The communication overhead is expected to be high.

As we are not using the Bazel build system as suggested, the adapted applications have dependencies on external libraries not part of the project. We link `libtensorflowlite_c`, `libtensorflow-lite` and their subdependencies from within the adjacent folder where it was cross compiled. The library `libedgetpu.so` compiled previously is manually copied to the project folder. For additional dependencies on system libraries not included in the cross-compilation environment (ie. `libdl`, `libpthread`, `libusb`, `libudev`), these are copied from the Coral platform into the project folder.

After compiling, the code is copied to the Coral for execution, along with its corresponding EdgeTPU model.

V. Results

A. Accuracy

Although accuracy is not a concern at this stage, we plotted the predicted vs expected results of each unquantized model as a sanity check, as shown in Fig. 2.

B. Execution Time

It was expected for the initial implementation that performance would be poor compared to the exact implementation due to the high communication overhead of passing a single vector per inference. Execution times for both cases are reported in Fig. 3.

Wildly varying results were obtained, with the TPU implementation taking between 7 and 700 times longer to execute than the exactly implementation running on the CPU. There appears to be no correlation in the increase of run time with either the length of the vectors or the complexity of the model.

C. Energy Consumption

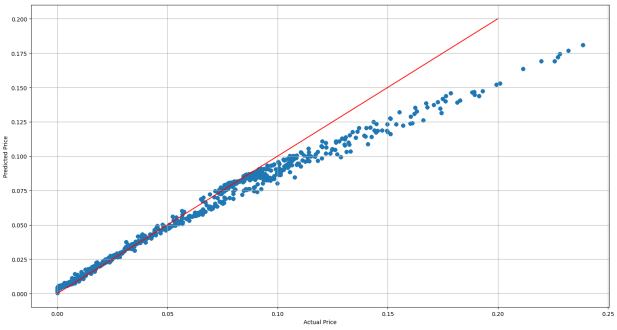
For initial testing, we intended to compare the energy consumption between the exact implementation and EdgeTPU implementation of each application using a Kill-A-Watt P4400 energy meter into which a USB power adapter is inserted to supply power to the Coral. Using this arrangement a power of approximately 3.9 W was drawn while the system was quiescent, rising to 4.8 W while processing models, with no difference between exact and TPU model evaluations.

Measurement of energy consumption was suspended when the unexpectedly long TPU processing times were noted, as it was suspected that results would be tainted by neither the CPU nor the TPU consistently consuming a majority of power during these times.

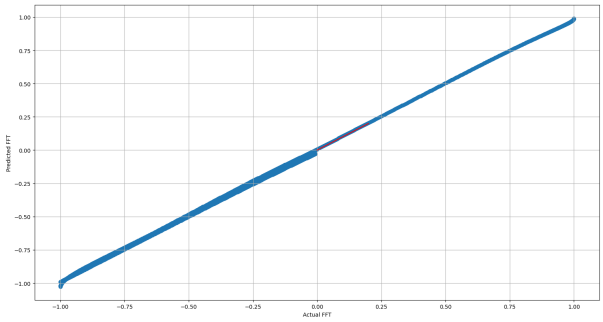
VI. Conclusions

A. Performance

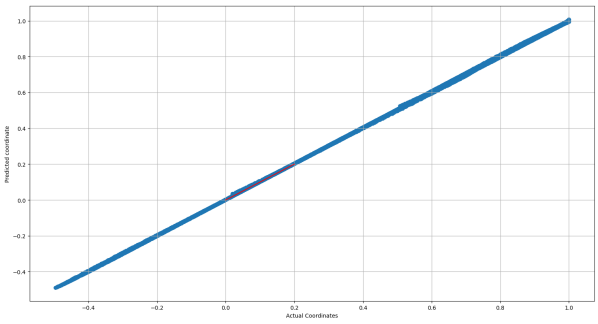
The performance of the model on the EdgeTPU was even more disappointing than expected for an initial attempt, taking



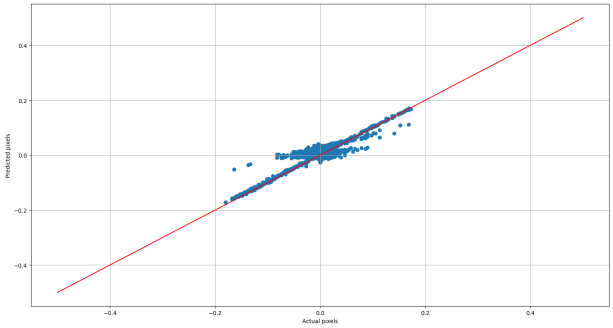
(a) Blacksholes case



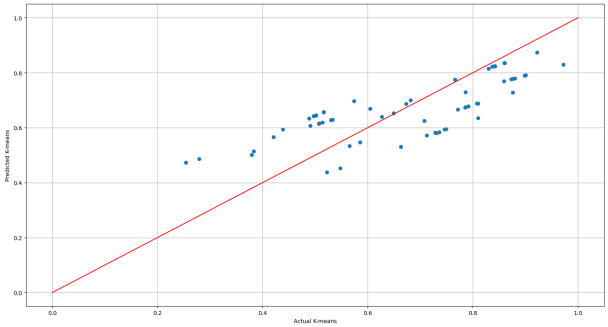
(b) FFT case



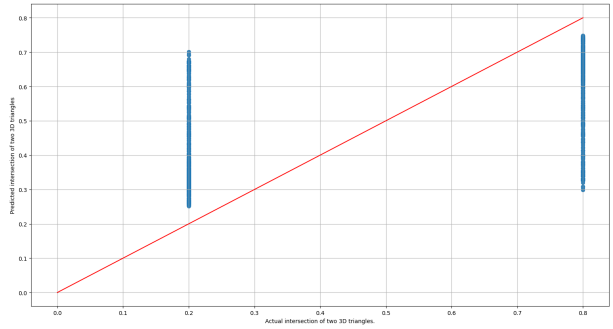
(c) Inversek2j case



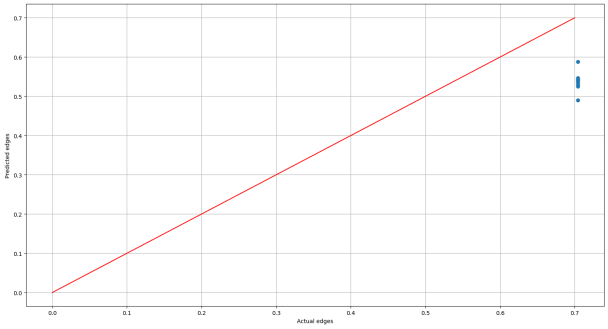
(d) Jpeg case



(e) kmeans case



(f) Jmeint case



(g) Sobel case

Fig. 2: Compare predictions to actual values.

Case	Topology	Exact Time	TPU Tme	Ratio
Blackscholes	6-16-16-1	1.82	81.57	44.9
FFT	1-4-4-2	0.68	212.61	311.0
Jmeint	18-32-8-2	27.35	442.64	16.2
Jpeg	64-16-64	10.51	70.66	6.7
Kmeans	6-8-4-1	2.64	1876.61	711.9
Sobel	9-8-1	27.12	334.87	12.3

Fig. 3: Exact / TPU Execution time comparison (seconds).

between 7 and 700 times longer to execute with no obvious correlation between the increase in execution time and any other factor. Because of this it was not possible to measure power or energy consumption with any accuracy.

In any case, energy measurements were crudely made, using the Kill-a-watt device to measure the consumption of the entire device over the execution of the entire application. A more sophisticated laboratory test would allow direct comparison of the energy consumption only during approximable regions of code.

B. Batched Transactions

For the initial phase of this work, we have made a simple substitution of the pragma-identified approximable regions, without algorithm modification. The authors of the Parrot transformation imagined its use with a tightly-coupled accelerator. However, for use with a loosely-coupled accelerator such as the EdgeTPU, this causes significant loss of performance due to greatly increased communication overhead and wait times. More effort is required to batch transactions so that they may be sent asynchronously en masse to improve performance, which may profoundly impact the observed execution time issues.

C. Training

Here we have applied post-training quantization to convert the models to an 8-bit integer representation suitable for the Edge TPU. This necessarily involves some loss of accuracy, and a possibly sub-optimal model topology. Although we are not yet preoccupied with accuracy at this stage, a better approach would be to perform quantization-aware training, perhaps with iterative testing of the models on the EdgeTPU to optimize their topology.

D. Software / Build Issues

We noted that TensorFlow is overwhelmingly a Python project, with limited support for other languages. In particular, the C/C++ API offers only minimal functionality, compared to the richness of the Python API. While this does not present difficulty for offline training operations, we feel that a production system with constrained resources would be less likely to use Python, and the limited APIs may provide some impediments.

The immature nature of the libraries used was at times an impediment to our work. Lack of version numbering, backward compatibility and binary compatibility between versions of `libedgetpu` extant "in the wild" frustrated us with their ability to cause segmentation faults, or failure to load versions of models with particular features. Recompilation the library is of course a solution, however this was also somewhat tedious when faced with a choice between a rigid and error-prone bazel build, an obsolete docker build, or a native build on an under-powered platform. In many cases we suffered "repository rot", where changes in the repositories of external dependencies (such as the case of the change in the URL of the Debian packages), or changes in the behavior in later versions of build tools (Bazel) caused packages to break. More stability would be desirable for use in a production system.

VII. Future Work

The observed execution time issue must be investigated and resolved, beginning with a profiling of execution on the EdgeTPU to identify the location of the bottleneck.

Possibly related to the execution time issue, algorithm changes are required to batch inference to reduce overhead and latency communicating with the accelerator.

Subsequent to batched inferences, improvements in the energy measurement methodology would permit finer grained measurement of energy consumption, particularly during approximable algorithm regions.

Following these improvements to the current project, the expected work in energy/accuracy tuning may begin, possibly including quantization-aware tuning.

VIII. Bibliographic References

References

[1] Nasrin Akbari and Mehdi Modarressi. "A High-Performance Network-on-Chip Topology for Neuromorphic Architectures". In: *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. Vol. 2. 2017, pp. 9–16. DOI: 10.1109/CSE-EUC.2017.188.

[2] dmitriykovalev. *Edge TPU runtime library (libedgetpu)*. https://github.com/google-coral/libedgetpu.

[3] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Dec. 2012. DOI: 10.1109/micro.2012.48.

[4] Isaias Felzmann, Joao Fabricio Filho, and Lucas Wanner. "Risk-5: Controlled Approximations for RISC-V". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020), pp. 4052–4063. DOI: 10.1109/tcad.2020.3012312.

[5] Tomás González-Aragón and Jorge Castro-Godínez. "Improving Performance of Error-Tolerant Applications: A Case Study of Approximations on an Off-the-Shelf Neural Accelerator". In: *2021 IEEE V Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI)*. 2021, pp. 1–6. DOI: 10.1109/JoCICI54528.2021.9794353.

[6] Google. *Dev Board datasheet — Coral — coral.ai*. https://coral.ai/docs/dev-board/datasheet/. 2022.

[7] Google. *Edge TPU*. URL: https://cloud.google.com/edge-tpu.

[8] Google. *Edge TPU Compiler*. https://coral.ai/docs/edgetpu/compiler/. 2020.

[9] Google. *Get started with the Dev Board*. https://coral.ai/docs/dev-board/get-started/. 2020.

[10] ACT Research Group. *AxBench*. https://bitbucket.org/act-lab/axbench/src/master/. 2015.

[11] Jie Han and Michael Orshansky. "Approximate computing: An emerging paradigm for energy-efficient design". In: *2013 18th IEEE European Test Symposium (ETS)*. IEEE, May 2013, pp. 1–6. DOI: 10.1109/ETS.2013.6569370.

[12] *Intel® Movidius™ Myriad™ X Vision Processing Unit (VPU) — intel.com*. https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu/movidius-myriad-x.html.

[13] *Intel® Neural Compute Stick 2 — intel.com*. https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html.

[14] Oliver Keszocze. "Approximate Computing". In: *it - Information Technology* 64.3 (May 2022), pp. 77–78. DOI: 10.1515/itit-2022-0027.

[15] Raspberry Pi Ltd. *Raspberry Pi 4 Model B specifications – Raspberry Pi — raspberrypi.com*. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[16] Thierry Moreau et al. "SNNAP: Approximate computing on programmable SoCs via neural acceleration". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 603–614. DOI: 10.1109/HPCA.2015.7056066.

[17] Steffen Nissen. *FANN Fast Artificial Neural Network Library*. URL: http://leenissen.dk/fann/wp/.

[18] Sourav Sanyal et al. "Approximate Computing for Machine Learning Workloads: A Circuits and Systems Perspective". In: *Approximate Computing*. Ed. by Weiqiang Liu and Fabrizio Lombardi. Cham: Springer International Publishing, 2022, pp. 365–395. ISBN: 978-3-030-98347-5. DOI: 10.1007/978-3-030-98347-5_15. URL: https://doi.org/10.1007/978-3-030-98347-5_15.

[19] Lukas Sekanina. "Introduction to approximate computing: Embedded tutorial". In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, Apr. 2016. DOI: 10.1109/ddecs.2016.7482460.

[20] Kiran Seshadri et al. "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks". In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 2022, pp. 79–91. DOI: 10.1109/IISWC55918.2022.00017.

[21] TensorFlow. *Build from source*. https://www.tensorflow.org/install/source.

[22] TensorFlow. *Build TensorFlow Lite with CMake*. https://www.tensorflow.org/lite/guide/build_cmake.

[23] vpetrigo. *arm-cmake-toolchains*. https://github.com/vpetrigo/arm-cmake-toolchains.

[24] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate Computing: A Survey". In: *IEEE Design & Test* 33.1 (2016), pp. 8–22. DOI: 10.1109/MDAT.2015.2505723.

[25] Amir Yazdanbakhsh et al. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing". In: *IEEE Design & Test* 34.2 (2017), pp. 60–68. DOI: 10.1109/MDAT.2016.2630270.