UltraTrail: A Configurable Ultra-Low Power TC-ResNet AI Accelerator for Efficient Keyword Spotting

Paul Palomero Bernardo, Christoph Gerum, Adrian Frischknecht, Konstantin Lübeck, and Oliver Bringmann, *Member, IEEE*

***Abstract*—Recent advances in machine learning show the superior behavior of temporal convolutional networks (TCNs) and especially their combination with residual networks (TC-ResNet) for intelligent sensor signal processing in comparison to classical CNNs and LSTMs. In this paper, we propose UltraTrail, a configurable, ultra-low power TC-ResNet AI accelerator for sensor signal processing and its application to efficient keyword spotting. Following a strict hardware/model co-design approach, we have derived an optimized low-power hardware architecture for generalized TC-ResNet topologies consisting of a configurable array of processing elements and a distributed memory with dynamic content re-allocation. We additionally extend the network with conditional computing to reduce the number of operations during inference and to provide the possibility for power-gating. The final accelerator implementation in Globalfoundries' 22FDX technology achieves a power consumption of 8.2 µW for the task of always-on keyword spotting meeting the real-time requirement of 100 ms per inference with an accuracy of 93 % on the Google Speech Command Dataset.**

***Index Terms*—Accelerator architectures, deep neural networks, edge computing, low-power electronics, neural network hardware.**

I. Introduction

THE important breakthroughs in using deep neural networks for a large variety of machine learning applications have been strongly influenced by the availability of high-performance computing platforms. In contrast to its biological origin, however, the high performance of artificial neural networks critically has much higher energy demands. Today's solution of applying modern deep learning approaches in mobile embedded systems very often relies on the necessity to access cloud computing platforms to process deep learning inference with the associated negative consequences for privacy and security. Therefore, advances are strongly needed both in the development of resource-efficient deep neural networks (DNNs) and their energy-efficient implementation in edge computing devices. In the last two years, temporal convolutional networks (TCNs) have shown their superior behavior for numerous sequence modeling and analysis tasks (cf. [1]) and in particular for analysis and classification of sensor data streams, just to mention an important application domain of embedded devices. TCNs have scalable temporal convolutional layers, cope with relatively flat hierarchies, and yet cover a large receptive field in a flexible and adaptive manner. Thus, they can significantly minimize the entire signal processing chain. Compared to LSTMs and GRUs, TCNs appears not only more accurate, but also simpler and clearer, and are therefore a very attractive candidate for efficient implementation of deep learning inference in edge computing devices.

The application of the TCN concept to the quite successful deep residual learning approach (ResNet [2]) has been proposed by Choi et al. in [3] and is called TC-ResNet. TC-ResNet uses convolutions that are only applied along the temporal dimension of the input data and allows detecting high-frequency features with less complex networks just by increasing the receptive field of the temporal convolutions.

The resource and energy-efficient implementation of deep neural networks, like TCNs, on mobile and embedded devices are also very sensitive to the underlying hardware architecture. There are already a great number of domain-specific hardware accelerators for efficient implementation of deep learning inference. Even though they usually claim to be domain-specific, the basic idea is always to provide a generic architecture to implement a broad range of deep neural networks. If we restrict ourselves to a well-suited subclass of DNNs, specifically TCNs, the accelerator design can be optimized much more effectively. Therefore, a configurable accelerator architecture is needed which allows for implementing a wide variety of temporal neural networks. However, a large number of parameters and operations combined with the frequently present real-time requirements pose significant challenges to the underlying accelerator architecture, especially under strict energy and area limitations found in near-sensor signal processing. A configurable hardware accelerator needs to be easily customized for a specific application and is crucial for tackling the task of real-time DNN inference.

In this paper, we propose UltraTrail, a configurable, ultra-low power TC-ResNet AI accelerator for sensor signal processing. UltraTrail combines highly parallel computing paradigms and a dataflow tailored to the TC-ResNet architecture with a distributed memory setup to exploit the data

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue. This work has been partly funded by the EU and the German Federal Ministry of Education and Research (BMBF) in the projects OCEAN12 (reference number: 16ESE0270) and Scale4Edge (reference number: 16ME0129).

The authors are with the Department of Computer Science, University of Tübingen, 72074 Tübingen, Germany (email: {paul.palomero-bernardo, christoph.gerum, adrian.frischknecht, konstantin.luebeck, oliver.bringmann}@uni-tuebingen.de).

dependencies within the temporal convolution and support the branched nature of the residual network. We present its application to efficient keyword spotting (KWS) using an automated training and model compression flow for mapping the model onto the hardware accelerator. Our contributions can be summarized as follows:

- Hardware-aware TC-ResNet implementation for efficient inference including weight/feature quantization and conditional computing (Section III).
- Configurable hardware architecture for ultra-low power TC-ResNet inference together with a cycle-accurate timing model (Section IV).
- Application of always-on KWS with a power consumption of 8.2 µW and an accuracy of 93 % on the Google Speech Command Dataset (GSCD) (Section V).

II. Related Work

One important technique for the efficient implementation of neural networks is the quantization of weights and activations to small bit widths [4]. Quantization can lead to enormous savings in memory requirements, the resource requirements of the arithmetic units, and the power consumption of the DNN accelerators [5]. Extreme quantizations like Ternary Weight Networks [6] or Binary Weight Networks [7] even allow the replacement of multiplication by addition or subtraction. If both activations and weights are quantized to 1 bit, multiplications can be replaced by element-wise binary operations [8].

Another idea to increase energy efficiency is the conditional computing of parts of a neural network. One of these techniques is early exiting of networks at exit branches [9] [10]. Here, a neural network gets extended by early exit branches near the front of the network. These exit branches are evaluated before the main branch of the network and if their prediction is correct with reasonable confidence, the rest of the network computations are skipped. A relatively new technique is the dynamic pruning of input channels at certain positions within the output of a convolutional layer (channel gating) [11]. In contrast to static pruning, channel gating allows a specific decision on the pruning of filter coefficients for each input. Energy efficiency can also be increased by hierarchical cascading systems [12]. There are several stages, each of which contains its own neural network and can perform increasingly complex classifications instead of using only one complex neural network. Thus, a successful classification can already be achieved in the first, non-complex stages. This prevents later more complex and thus more expensive stages in terms of energy efficiency from being executed. So they can be switched off in the meantime. Our implementation of the tooling to train and deploy efficient neural networks builds on existing work for neural network compression and conditional execution to support an energy-efficient implementation on the hardware accelerator.

As the use-case for the hardware accelerator in this paper is a keyword spotting task, we also compare our results to dedicated hardware accelerators for keyword spotting. Previous efforts in this area have focused on the execution of established network architectures. Price et al. [13] presented a

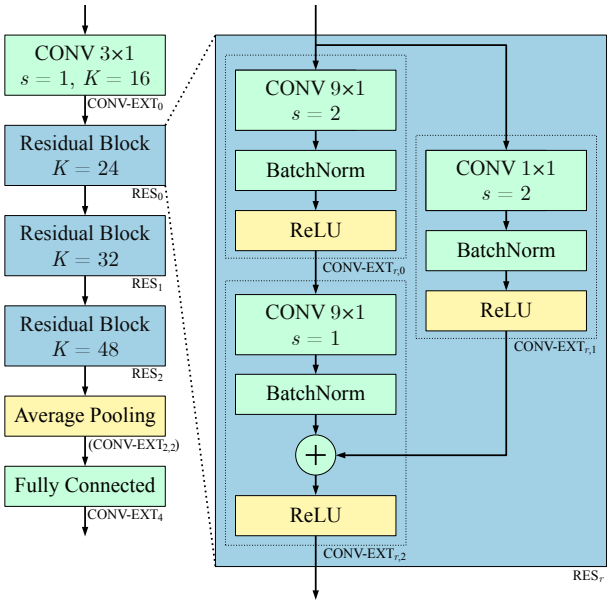

Fig. 1. Example residual network for temporal convolution (tc-res8).

scalable circuit architecture for automatic speech recognition with deep feed-forward networks. Liu et al. [14] introduced an ultra-low power KWS accelerator using a 2D convolutional neural network (CNN) and Giraldo et al. [15] employed an ultra-low power long short-term memory (LSTM) accelerator for the task of KWS in their speech-triggered wake-up SoC Vocell. However, with the continuous advances in machine learning new network architectures based on temporal convolution (i.e., 1D convolution along the temporal dimension) have shown superior behavior for intelligent sensor signal processing. Recently, Choi et al. [3] introduced the TC-ResNet, a combination of temporal convolution and residual networks for real-time KWS, which considerably reduces the number of operations compared to conventional 2D convolutional and recurrent network architectures while maintaining comparable accuracy. To the best of our knowledge, there is no efficient hardware implementation of TC-ResNet based neural networks. In the next section, we describe our approach for target-specific optimization of the neural network model. Our configurable hardware accelerator is presented in section IV and the achieved results for KWS in section V.

III. Model Optimization and Quantization

The TC-ResNets considered in this paper implement the basic structure described in [3], where TC stands for temporal convolutions and ResNet [2] adopts the ResNet approach that is most well-known for image recognition and classification tasks. The most notable difference between normal ResNets and TC-ResNets is the use of 1D-convolutions. These convolutions are only applied along one dimension of input data. In the case of time series data, this is the temporal dimension. One advantage of temporal convolutional networks is their use of larger convolution filters with the same number of parameters and arithmetic operations as compared to traditional convolutional networks using 2D-Convolutions, e.g. 9x1 vs 3x3. This

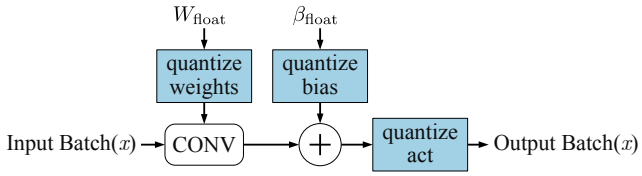

Fig. 2. Quantization-aware training.

allows them to detect high-frequency features with shallower networks and makes them one of the best emerging network technologies for the analysis of 1D time series, e.g. taken from sensor data streams.

A TC-ResNet as depicted on the left of Fig. 1 consists of a feature extraction part composed of a sequence of 1D-convolution and residual blocks and a classifier part composed of a global average pooling and a small fully connected layer.

A residual block as shown on the right of Fig. 1 consists of a main path with the following layers: 9x1-convolutional layer $\rightarrow$ batch normalization $\rightarrow$ ReLU $\rightarrow$ 9x1-convolutional layer $\rightarrow$ batch normalization. The outputs of the main path are then added to the residual path. The residual path forwards the input data of the residual block. The addition is followed by a final ReLU activation function. The residual block usually contains a simple 1x1-convolutional layer followed by batch normalization and ReLU for resampling the inputs of the block to the output size.

Our model selection and preparation framework tries to select a TC-ResNet optimized for low-power inference on the target hardware architecture. It consists of the following steps.

1) *Neural network architecture exploration*: This step modifies the parameters of the basic TC-ResNet architecture to search for trade-off between model complexity and accuracy. Architecture exploration is carried out with unquantized networks in floating point arithmetic to speed up the exploration.
2) *Quantization-aware training*: This step takes the selected model and retrains it with simulated quantization of the networks parameters and activations.
3) *Conditional execution*: Finally, the network is transformed by inserting early exit branches to conditionally avoid the computation of parts of the network on most of the samples.

A. Quantization-Aware Training

Many approaches to neural network quantization [4] take a neural network trained on floating-point data and simply quantize the weights to integers. This approach leads to sufficient accuracy for reasonably deep networks as these networks provide enough redundancy to handle the loss of arithmetic accuracy. As TC-ResNets are designed as shallow networks with a relatively low amount of redundancy, we need to adopt a training approach that takes the effects of neural network quantization into account.

For this, we use the approach shown in Fig. 2. Weights, biases, and activations are quantized using a simulated quantization function. In this paper, we use a simple symmetric fixed-point quantization. A floating-point value v_{float} is transferred

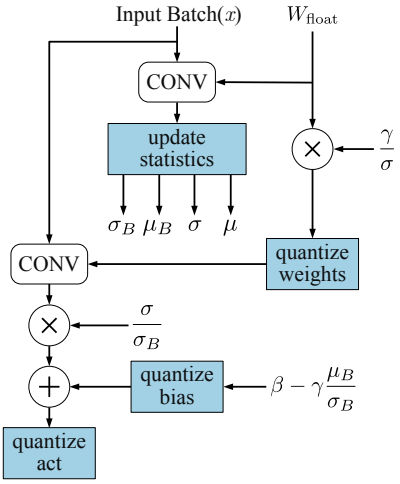

Fig. 3. Quantization-aware batch norm folding.

to its simulated n-bit quantized representation v_q using the following equation:

$$v_q = \frac{\max(\min(\text{round}(v_{float}) \cdot 2^{n-1}), 2^{n-1}-1), -2^{n-1})}{2^{n-1}}. \quad (1)$$

At inference time on the hardware accelerator, we use correctly quantized fixed-point values. Quantization effects are only simulated during the forward phase of the training. During backpropagation we adopt the straight through estimation approach [16], [17], replacing quantization functions with identity functions and updating the floating-point values of weights directly. This allows small updates of weights to accumulate over multiple mini-batches even if a single update would be too small to affect the quantized weights.

For further simplification of the hardware implementation, all division operations in the neural network are replaced by an approximation, rounding each divisor to the nearest power of 2. This approach allows a simple implementation of the division operations using only a single right-shift operation. This is especially used in the average pooling layers and in the calculation of early exit conditions as shown in subsection IV-B4.

During quantization-aware training, special care must be taken for batch normalization. Batch normalization is an important operation at training time. It normalizes the outputs of the preceding layers and allows a faster convergence of the training process and a higher classification accuracy of the trained network.

Batch normalization is defined by the following operation:

$$y_i = \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B + \epsilon}} + \beta, \quad (2)$$

with μ_B and σ_B denoting the per mini-batch mean and standard deviation of the input activations. γ and β are learned parameters, and epsilon is a small constant to increase numerical stability for small σ_B. During inference batch normalization is calculated as:

$$y_i = \gamma \frac{x_i - \mu}{\sqrt{\sigma + \epsilon}} + \beta, \quad (3)$$

with μ and σ denoting the long-term moving averages of inputs over the training set. As all parameters of this equation except the input x_i are constant at inference time, these parameters can be merged with the preceding input quantization layers. This optimization is often referred to as batch norm folding [4].

For floating-point models, this transformation can be implemented as a simple post-training transformation. This transformation uses the following transformation to calculate the new weights and biases of the preceding convolutional or fully connected network layer.

$$W_{\text{inf}} = \gamma \frac{W}{\sqrt{\sigma + \epsilon}} \text{ and } \beta_{\text{inf}} = \beta - \gamma \frac{W}{\sqrt{\sigma + \epsilon}}. \tag{4}$$

The batch normalization layer is then removed from the network. But for low-bitwidth quantized models, the effects of this optimization need to be considered during network training. Otherwise, the rounding errors that are produced by batch norm folding. would severely degrade the accuracy of the neural network. This is implemented according to Fig. 3.

We first calculate a convolution with the floating-point weights. The result of this convolution is then used to update the per batch and global statistics. The floating-point weights are used to calculate the convolution weights after batch norm folding and quantization. The actual convolution uses the quantized weights with simulated batch normalization. After the convolution, we first correct the result for the usage of global statistics instead of per batch statistics and finally add the folded biases. The outputs of the folded layer are then quantized with the output quantization function of the convolutional layer.

During inference, only the convolution with quantized weights and the addition of the quantized bias are calculated. The rest of the operations are removed from the network.

B. Conditional Execution

Our implementation of conditional execution for temporal convolutional models builds on [9], [10].

It works by inserting additional exit branches after the first one or two residual blocks, as shown in Fig. 4. During training, the main branch and each exit of the network are always executed and trained simultaneously. During inference, the exit branches are evaluated first, and if an exit criterion is met the execution stops at the exit branch and returns the output of the current exit as the classification result of the whole network. If the exit is not taken, the rest of the network is evaluated until either one of the exits is taken or the normal exit of the network is reached.

Our exit criterion is based on the cross-entropy loss-function used to train the networks. This function is defined as follows:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right), \tag{5}$$

where x is the vector of outputs of the neural network and $class$ is the expected class number. During training the expected class is given in the training data, to calculate an approximation of the loss function during inference we replace

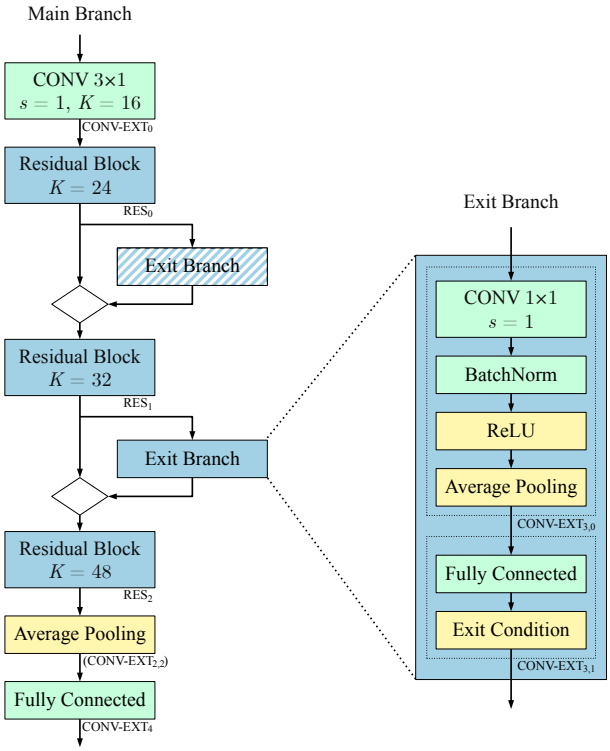

Fig. 4. Implementation of early exit.

$class$ with the index of the maximum output of the exit. This essentially calculates the exit condition assuming that each output always makes the correct decision, and using the loss as a confidence value for the correctness of this decision. This leads to the following exit condition:

$$-\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) < T. \tag{6}$$

Using a user-defined threshold T. This equation can be transformed to

$$\sum_j \exp(x[j] - x[class]) \;<\; exp(T). \tag{7}$$

During quantization aware training, we implement the exit condition using the approximation for exp described in section IV-B4.

IV. Hardware Architecture

The hardware architecture for UltraTrail aims to provide efficient mapping of the TC-ResNet without giving up too much of its flexibility. To this end, we introduce a unified layer representation called CONV-EXT. A CONV-EXT layer is a CONV layer which can be extended with batch normalization, ReLU activation, average pooling, and conditional computing in this order. Coupling these operations to a CONV layer and enforcing a predefined execution order makes it possible to streamline the dataflow within the accelerator while maintaining most flexibility since this is generally the preferred processing order within the TC-ResNet.

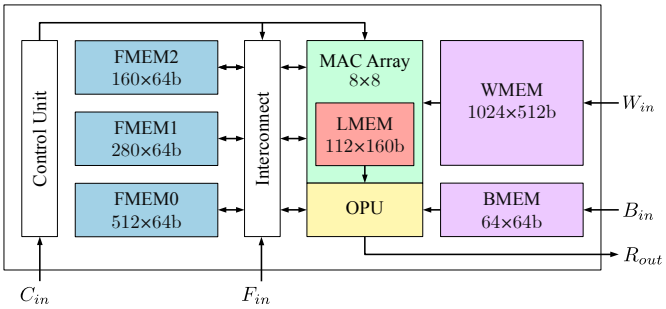

Fig. 5. Overview of the UltraTrail system architecture.

Based on this concept, UltraTrail can execute networks with up to 16 CONV-EXT layers. The complete system architecture is shown in Fig. 5. It employs a MAC array for the convolutional computation and an output processing unit (OPU) to perform the remaining post-processing steps. They are surrounded by a distributed memory setup consisting of a weight memory (WMEM), bias memory (BMEM), and three feature memories (FMEM0-2) to meet the high bandwidth requirements and support the branched network structure introduced by the residual connection and conditional computing. To support changes in the network structure, configurability is given through means of a programmable control unit.

A. MAC Array

The MAC array forms the primary processing unit of the accelerator and is used to execute the computationally intensive yet highly parallelizable CONV layers. A detailed schematic view is presented in Fig. 6. It consists of 64 MAC units arranged as an 8×8 grid as well as a local memory (LMEM) to store partial sums. The dataflow is defined by spatially unrolling input and output channels along the two array dimensions. Both size and dataflow are fit for our optimized TC-ResNet. Since the number eight is the greatest common divisor for all input and output channels (except for the final output), almost optimal utilization of the available processing units can be achieved by using this MAC array size.

A CONV/FC layer is processed iteratively using the layers' spatially unrolled convolutional loop nest. A general version of a spatially unrolled convolutional loop nest with output channels K, input channels C, filter width F, output channel width X, a stride of one, no zero-padding and input, weight and output tensors i,w and o, is described in Fig. 7. Note, that fully-connected (FC) layers can be described similarly by setting $F = X = 1$. In each cycle, 64 weights $(w_{00}, \ldots, w_{77})$ from the WMEM and 8 input features $(i_0, \ldots, i_7)$ from the corresponding FMEM are provided to the array. The memories are directly connected to the array via their output ports and provide the quantized data in vectorized form. During the design phase, the bit widths for weights and input features can be adjusted as required. Our final accelerator uses 6-bit weights and 8-bit input features, so the port widths of the memories are fixed to $64 \times 6\,\text{bit} = 384\,\text{bit}$ (WMEM) and $8 \times 8\,\text{bit} = 64\,\text{bit}$ (FMEM). Which data is provided at what

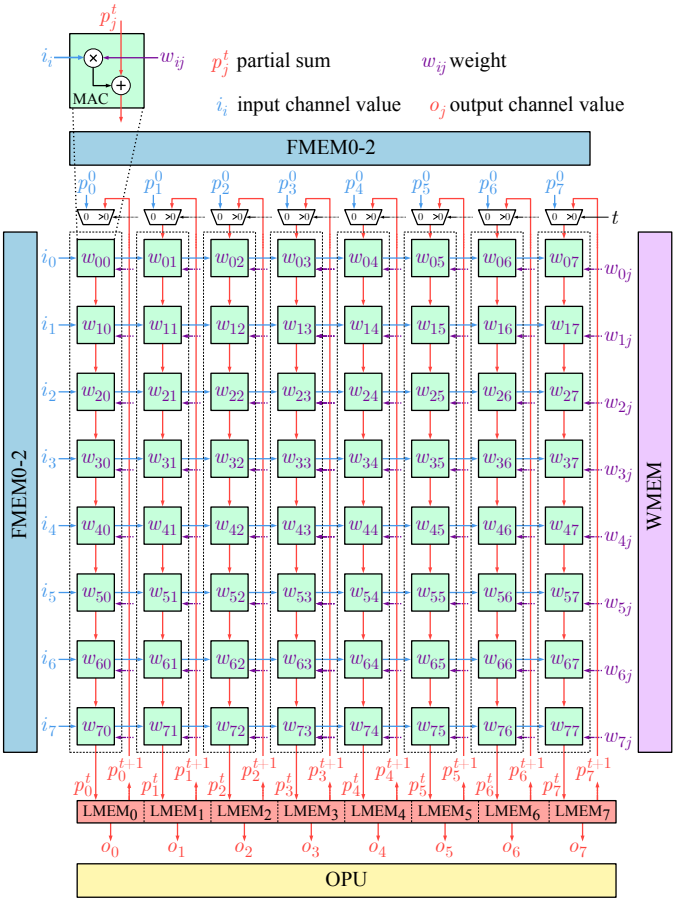

Fig. 6. Schematic view of the MAC array.

point in time results from the loop body of the convolutional loop nest. The eight input features are taken from adjacent input channels and are broadcast along the output channel dimension to the individual MAC units which each receives the corresponding weight from one of eight filters. Thus, the array always processes eight input channels for eight output channels in parallel. The MAC units of each output channel form a combinational path along which the partial sum of an output feature is accumulated. Each MAC unit forms the product of weight and input feature and adds it to the incoming partial sum. To avoid overflow or loss of accuracy, the bit width of the partial sums is 20 bit. The first MAC unit on the path generally receives the partial sum of a previous iteration from the LMEM. Only in the first iteration of an output feature ($c_0 = f = 0$) it is externally set to either 0 or an initial value $(p_0^0, \ldots, p_7^0)$. The latter is used to add two feature maps by initializing the partial sum to the corresponding feature value from the already computed feature map. All other units receive the result of the previous unit as input. The complete partial sum for the current time step t $(p_0^t, \ldots, p_7^t)$ is finally stored in the LMEM at the next positive clock edge. Simultaneously, the next partial sums $(p_0^{t+1}, \ldots, p_7^{t+1})$ are already loaded. Access to the LMEM is, similar to the other memories, vectorized using $8 \times 20\,\text{bit} = 160\,\text{bit}$ ports. To enable parallel read and write access, the LMEM is implemented as a dual-port memory. When the last iteration of the current output features

```
for k0 = 0 : K/8 do
    for c0 = 0 : C/8 do
        for f = 0 : F do
            for x = 0 : X do
                parfor k1 = 0 : 8 do
                parfor c1 = 0 : 8 do
                    o[8·k0 + k1][x] +=
                    i[8·c0 + c1][x + f]·
                    w[8·k0 + k1][8·c0 + c1][f]
```

Fig. 7. Spatially unrolled convolutional loop nest.

is reached ($c_0 = C/8, f = F$), i.e., they are completely accumulated, they are no longer stored in the LMEM but are forwarded to the OPU in the same clock cycle ($o_0, \ldots, o_7$).

This dataflow has two major advantages: First, due to the chosen loop-blocking scheme weights remain stationary to a MAC unit. This means that the same weights are used in consecutive cycles until they have been used in all required computations. As a result, the number of WMEM accesses is minimized since each weight is fetched only once throughout an entire inference. Second, as long as the number of input and output channels of a CONV layer is divisible by eight, an optimal operation mapping in terms of MAC utilization is possible which enables a continuous inference computation without stalls at a constant and maximal throughput of 64 MACS/cycle.

B. Output Processing Unit (OPU)

The OPU combines bias, ReLU activation, average pooling, and conditional computing and thus all remaining operations of the TC-ResNet in a combinational circuit with four discrete stages. It receives the output features generated by the MAC array and passes them through the entire circuit within the same clock cycle. Stages that are not required are skipped according to the current layer configuration. The results are written back to a feature memory. This immediate processing of the output features avoids redundant memory accesses, improves MAC array utilization, and streamlines the control flow inside the accelerator as it boils down to sequential execution of CONV layers. The individual stages are briefly described below.

1) Bias: The first stage adds a bias to the output features. It is directly connected to the BMEM from which it receives the biases in a vectorized form via an $8 \times 8\,\mathrm{bit} = 64\,\mathrm{bit}$ port. As for the weights and input features, the bit width for the biases can be adapted during the design phase. Although the TC-ResNet does generally not use biases within the CONV layers, this stage is necessary to support the bias introduced by folding the batch normalization as described in section III-A.

2) ReLU: The second stage applies a ReLU activation to the output features.

3) Average Pooling: The third stage performs an online average pooling for each output channel. For each channel, the incoming features are divided by the number of features per channel and are added onto a locally stored running mean. The number of features is approximated by the next larger power of two to implement the division by a right shift. Once a set of output channels has been finished, the result is forwarded and the local registers are reset.

4) Conditional Computing: The fourth stage computes the exit condition presented in (7) using a third-degree Taylor approximation around 0. Divide operations are approximated by right shifts using the next larger power of two. If the condition is met, the accelerator terminates inference and returns the current result. Unlike the previous stages, the calculation cannot be performed online since the Taylor approximation requires the maximum over all output features. Incoming features are, therefore, stored locally until the entire output is available.

TABLE I
FMEM ARRANGEMENT FOR OUR TC-RESNET KWS MODEL

	FMEM0	FMEM1	FMEM2
CONV-EXT$_0$	IF	OF	–
CONV-EXT$_{0,0}$	OF	IF	–
CONV-EXT$_{0,1}$	–	IF	OF
CONV-EXT$_{0,2}$	IF	OF	PS
CONV-EXT$_{1,0}$	OF	IF	–
CONV-EXT$_{1,1}$	–	IF	OF
CONV-EXT$_{1,2}$	IF	OF	PS
CONV-EXT$_{3,0}$	–	IF	OF
CONV-EXT$_{3,1}$	–	–	IF
CONV-EXT$_{2,0}$	OF	IF	–
CONV-EXT$_{2,1}$	–	IF	OF
CONV-EXT$_{2,2}$	IF	OF	PS
CONV-EXT$_4$	–	IF	–

C. Distributed Memory Setup

A distributed memory setup with dynamic content reallocation is used to account for the high-bandwidth requirements and parallel read/write operations of the MAC array and the OPU. It consists of a weight memory (WMEM), a bias memory (BMEM), three feature memories (FMEM0-2), and the local memory (LMEM) inside the MAC array.

1) WMEM and BMEM: These two memories contain the entire network model. Once initialized, they remain unchanged during further operation. The WMEM has a size of 64 kB and is by far the largest memory in the accelerator.

2) FMEM: The three feature map memories store the input and output feature maps and are switched dynamically to exploit the fact that the output feature map of the previous layer is the input feature map of the next layer. For instance, the initial feature map is loaded into FMEM0 and the output feature map of the first layer is stored into FMEM1. For the next layer, FMEM1 will provide the input feature map and the generated output feature map is stored back into FMEM0. However, this straight forward switching is disrupted by the ResNet structure as well as the exit branch where an additional feature map for the parallel path needs to be managed. For this purpose, FMEM2 is introduced which stores this extra feature map. In the case of a residual connection, it also provides the features through partial sum initialization to seamlessly integrate the addition of both feature maps into the dataflow.

TABLE I summarizes the FMEM arrangement for our entire KWS model. Each memory is either providing input features (IF), partial sums (PS), receives output features (OF), or is idle (–). Note, that all feature memories are generally interchangeable and can perform every role. The FMEM arrangement for a layer is defined in the layer configuration.

3) LMEM: The LMEM is used to decouple the high precision (20 bit) domain within the MAC array from the other feature memories (8 bit). Even though it would generally be possible to remove the LMEM and directly accumulate the partial sums inside an FMEM, this would introduce a huge overhead as each FMEM would need to support the 20-bit feature representation.

D. Control Unit

UltraTrail employs a programmable control unit to execute networks with up to 16 CONV layers including their post-processing steps. A 672-bit configuration register allows the network structure to be described on a layer-by-layer basis. For each layer, the loop boundaries of the convolutional loop nest, stride, and padding, as well as the post-processing steps to be performed and the arrangement of the feature memories are defined. All control signals can then be derived from these values and the loop variables of the convolutional loop nest which are updated by every clock cycle.

E. Timing

For a design-space exploration of neural networks to be used in real-time systems it is crucial to know how long the inference of a sample takes to verify given real-time constraints before the neural network is deployed onto the target hardware. Thus, we propose an analytical timing model for UltraTrail to predict the cycle-accurate run time for one inference.

Each CONV-EXT layer to be mapped onto UltraTrail can be defined as a 6-tuple[1] $l = (C, C_w, K, F, s, p)$, where:

- $C \in \{1, 2, 3, \ldots, 56\}$ is the number of input channels;
- $C_w \in \{1, 2, 3, \ldots, 127\}$ is the width of an input channel;
- $K \in \{1, 2, 3, \ldots, 56\}$ is the number of output channels;
- $F \in \{1, 2, 3, \ldots, 15\}$ is the filter width;
- $s \in \{2^n : n \in \{0, 1, 2, \ldots, 7\}\}$ is the stride;
- $p \in \{0, 1\}$ is the padding ($p = 1$ means padding enabled).

A neural network can be expressed as a set $\mathbb{L}$, which holds all network layers l.

The following equations describe how the number of clock cycles to process a whole neural network on the UltraTrail architecture is calculated. The approach is illustrated by repeated application of a generalized filter onto a single input channel as depicted in Fig. 8. Depending on the padding p, a filter is applied onto an input channel within the range of

$$\hat{C}_w = C_w + p \cdot 2 \cdot \left\lfloor \frac{F}{2} \right\rfloor. \tag{8}$$

[1]OPU computations are excluded from this tuple since they do not influence the timing due to their combinational implementation.

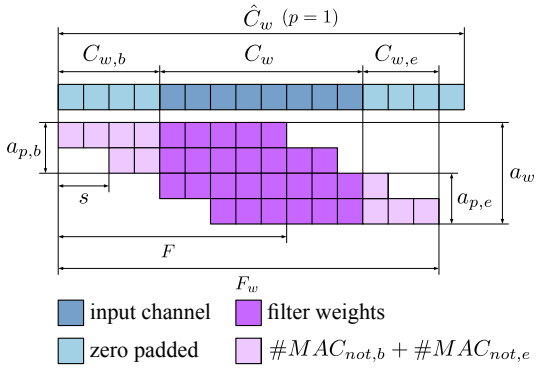

Fig. 8. Depiction of generalized filter applications onto a single input channel.

Within the range of $\hat{C}_w$, the whole filter is applied

$$a_w = \left\lfloor \frac{\hat{C}_w - F}{s} + 1 \right\rfloor \tag{9}$$

times. When padding is enabled ($p = 1$), zero padding is applied such that certain filter elements at the beginning and the end of an input channel are not used. The number of zero padded values at the beginning $C_{w,b}$ is defined by

$$C_{w,b} = \left\lfloor \frac{F}{2} \right\rfloor. \tag{10}$$

For the zero padded values, less than F MAC operations are executed by $a_{p,b}$ filter applications, which is calculated by

$$a_{p,b} = p \cdot \left\lfloor \frac{C_{w,b} - 1}{s} + 1 \right\rfloor. \tag{11}$$

Based on the number of filter applications, the number of MAC operations at the beginning of an input channel is defined by:

$$\#MAC_{not,b} = \sum_{i=0}^{a_{p,b}-1} \left\lfloor \frac{F}{2} \right\rfloor - s \cdot i. \tag{12}$$

The width F_w in which a filter is actually applied depend on the filter width F and the stride s is calculated by

$$F_w = a_w \cdot s + F - s \leq \hat{C}_w. \tag{13}$$

Since $F_w \leq \hat{C}_w$ holds, the width in which the filter is applied at the end of an input channel can be computed by

$$C_{w,e} = F_w - C_w - C_{w,b}. \tag{14}$$

The number of filter applications at the end of an input channel $a_{p,e}$ which include zero padding is given by

$$a_{p,e} = p \cdot \left\lfloor \frac{C_{w,e} - 1}{s} + 1 \right\rfloor. \tag{15}$$

Hence, the number of MAC operations being replaced by zeros which do not require a MAC unit can be calculated by

$$\#MAC_{not,e} = \sum_{i=0}^{a_{p,e}-1} \left\lfloor \frac{F}{2} \right\rfloor - s \cdot i - (C_{w,b} - C_{w,e}). \tag{16}$$

For each layer $l \in \mathbb{L}$, it takes one clock cycle to load the first input channel values and filter weights from FMEM0 and WMEM into the MAC array. All following loads are hidden

since loading the next input channel values and weights takes place in the same clock cycle as the processing of the current input channel values and filter weights. For the calculation of each output channel all input channels are needed which leads to $C \cdot K$ iterations. The MAC array processes eight input data for eight output channels on 64 MAC units in parallel such that the number of iterations is reduced to $\lceil \frac{C}{8} \rceil \cdot \lceil \frac{K}{8} \rceil$. During each iteration, the filter is applied a_w times onto one input channel and in each application F MAC operations are executed. When applying a filter onto one input channel, the zero-padded values at the beginning and the end of the input channel do not occupy a MAC unit and can, therefore, be subtracted from the number of MAC operations. This results in

$$t(l) = 1 + \left\lceil \frac{C}{8} \right\rceil \cdot \left\lceil \frac{K}{8} \right\rceil \cdot (a_w \cdot F - \#MAC_{not,b} - \#MAC_{not,e}) \tag{17}$$

clock cycles to process one CONV-EXT layer $l \in \mathbb{L}$. The total number of clock cycles to process a whole network is calculated by

$$T(\mathbb{L}) = \sum_{l \in \mathbb{L}} t(l). \tag{18}$$

V. Results

To evaluate our hardware accelerator and our neural architecture optimization framework, we show how to optimize the TC-ResNet for keyword spotting and its efficient implementation on our hardware accelerator.

In the remainder of this section, we first describe the results of the software-based neural network training using our implementation flow followed by an evaluation of the resulting network on the hardware accelerator and a comparison to the state of the art.

A. Network Training and Optimization

This section describes the results of the neural network training and optimization.

Our neural networks have been implemented in *pytorch* version 1.1.0 [18]. The model transformations have been implemented using *Nervana Distiller* [19] .

The networks are trained to classify the 10 keywords "yes", "no", "up", "down", "left", "right", "on", "off", "stop", and "go" from *Google Speech Commands* Dataset [20]. The remaining keywords are mapped to a common class "unknown", and samples containing only background noise are mapped to a final class "silence". This leads to 12 classes in total for our classification problem. The audio files of the dataset have a length of one second and are first preprocessed by shifting their position randomly 100 ms forward or backward and padding the remainder of the files with zeros. This results in a real-time constraint of 100 ms per inference to detect each keyword. After this data augmentation, a random amount of background noise taken from the noise files provided by Google Speech Commands is added to the sample. The intensity of the added noise is chosen randomly between 0.0 and 0.1 If not stated otherwise the added noise is used during training, validation and test.

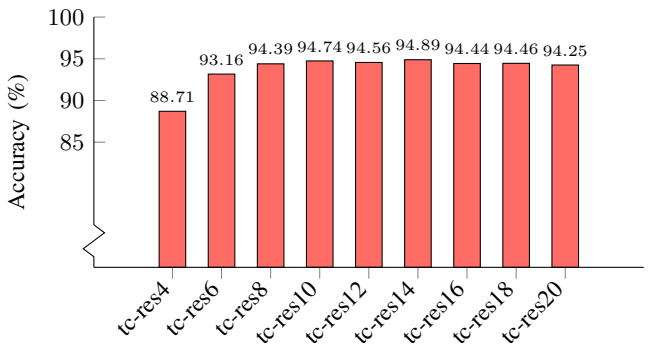

Fig. 9. Accuracies of different TC-ResNet architectures.

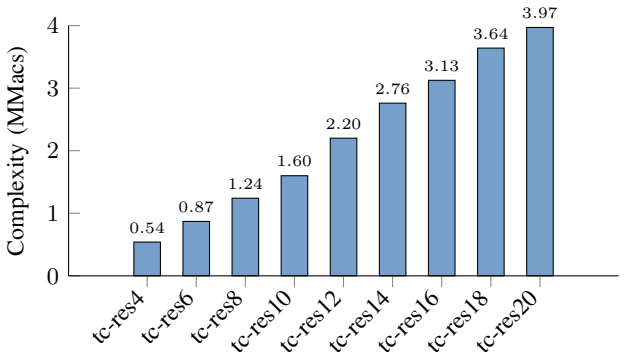

Fig. 10. Computational complexity of different TC-ResNet architectures.

For feature extraction, forty-dimensional mel-frequency cepstrum coefficient (MFCC) frames are constructed for each 30 ms window with a shift of 10 ms between each window. All frames are then concatenated to form the 40 channel input stream for our models with a length of 101 frames per second.

All networks are trained for 500 epochs using an initial learning rate of 0.1. The learning rate is reduced by a factor of 10 every 100 epochs. All other training parameters are left at their default values.

1) Neural Network Architectures: The results of our baseline network exploration are shown in Fig. 9 and Fig. 10. The computational complexity of the different TC-ResNet architectures increases linearly with the number of input layers. The accuracy of the networks reaches a saturation point around *tc-res8*. Therefore, we have chosen tc-res8 as the baseline architecture for the implementation of the keyword spotting system.

2) Quantization-Aware Training: After the selection of the initial network topology, we quantized the network. In this case, the whole network was retrained using quantization-aware training with user-selectable bit widths for weights w and activations a. The accuracy results are shown in Fig. 11. As the results for $a = 8$, $a = 6$ and $w = 4$ are clearly inferior to the rest of the results, which are in the same order of magnitude, we chose $a = 8$ bit and $w = 6$ bit as activations for the hardware accelerator.

3) Conditional Execution: To further reduce the average computational complexity of our neural networks, we integrated early exit branches after the first and second residual

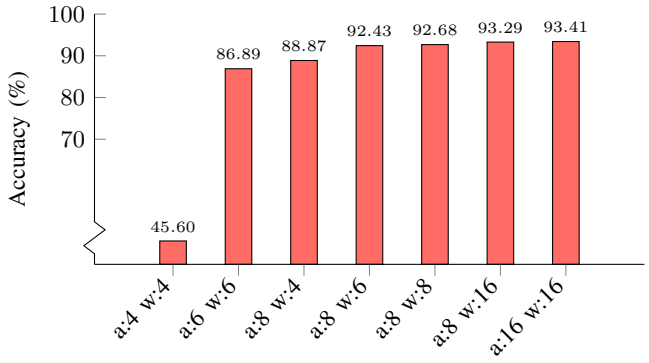

Fig. 11. Effect of varying bit widths for weights (w) and activations (a) on accuracy.

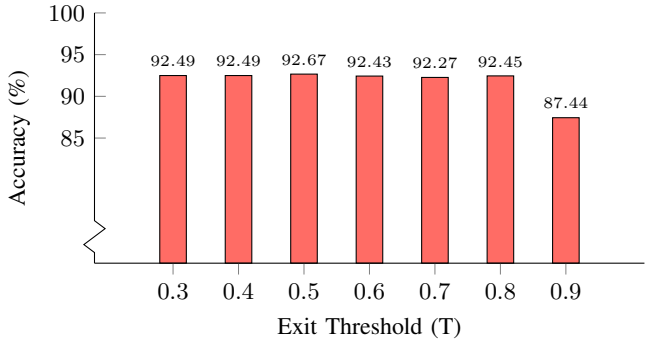

Fig. 12. Accuracy of early exit network tc-res8 with early exits after the first and second residual block under varying thresholds.

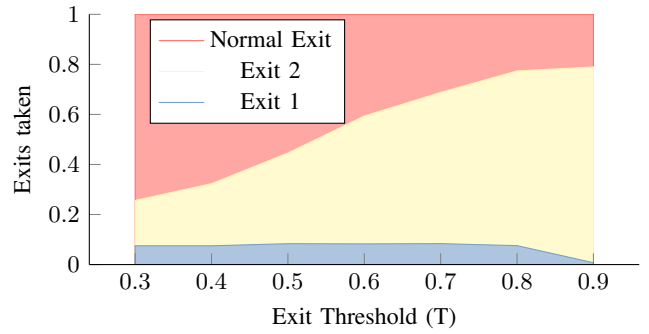

Fig. 13. Distribution of exits taken (exit 1, exit 2, normal exit) depending on a given exit threshold using quantized TC-ResNet8.

block of the tc-res8 network and evaluated the resulting networks. Both exit branches use the same thresholds. All networks in this section use tc-res8 topology and quantization of 8 bit for activations and 6 bit for weights. As can be seen in Fig. 12 the accuracy varies between 92.67% and 87.44% for a threshold of 0.5 and 0, respectively. The computational complexity, as shown in Fig. 13, varies drastically depending on the threshold value. While a threshold of 0.3 takes the first exit in 7% and the second exit in 18% of the samples only, the second exit is taken for more than 69% at a threshold of 0.8. The first exit on the other hand is still taken only 7% of the time. Therefore, we selected the final network to use a threshold of 0.8. As the first exit is only taken in very few cases, the final network for the hardware accelerator only contains the second exit.

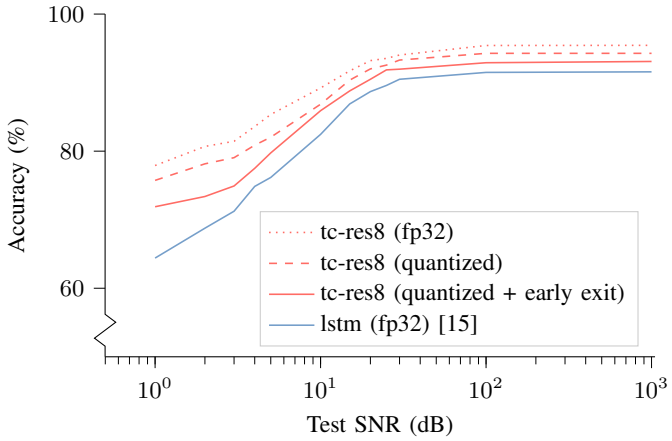

Fig. 14. Behavior in noisy environments compared to the network implemented on a state of the art accelerator [15].

4) Behavior in Noisy Environments: In Fig. 14 we evaluated our tc-res8 based networks under varying amounts of background noise. Taken from the noise files provided by the Google Speech Commands dataset. The tc-res8 network with early exit, using an exit threshold of 0.8 and quantized to 6-bit weights and 8-bit activations, reaches an accuracy of 93.09 % on an almost clean test set (SNR: 1000 dB), while still providing reasonably good accuracy up to an SNR of 20 dB (90.49 %).

To allow a better comparison with the state of the art we also included a 32 bit floating-point variant of the neural network used in [15]. As our implementation of this network uses floating-point instead of fixed-point values, this implementation reaches a slightly higher accuracy of 91.57 % compared to the 90.87 % reported in [15]. This network still performs worse than our tc-res8 variants over the whole range of added noise and especially when using very high noise levels the tc-res8 variants show an improved noise resistance.

B. Hardware Implementation

We implemented UltraTrail using the 22FDX platform by Globalfoundries. A die layout measuring 0.45 mm×0.45 mm is shown in Fig. 15. To minimize the static power consumption, low-leakage standard cells and memories are used, which are, by design, fixed to an operating voltage of 0.8 V. The chip supports clock frequencies of up to 30 MHz. For this implementation, a clock frequency of 250 kHz is chosen such that the real-time requirement of 100 ms per inference as described in section V-A is just met. During operation, clock gating is applied to idle memories to reduce their dynamic power consumption. In between inferences, they are additionally power gated.

The average power consumption of the layout during operation is evaluated through a time-based analysis of inference simulation traces at a 25 °C TT corner. Each trace consists of an initial setup phase followed by 20 consecutive inference computations. The results of two traces, one with and one

TABLE II
RESULTS COMPARISON

	ISSCC'2017 [13]	IEEE Access [14]	ISSCC'2020 [15]	ESSCIRC'2018 [21]	This work
Technology	65 nm	22 nm	65 nm	65 nm	22 nm
Area	13.17 mm^2	0.75 mm^2	2.56 mm^2	1.03 mm^2	0.20 mm^2
Frequency	3 MHz	250 kHz	250 kHz	250 kHz	250 kHz
Latency	-	20 ms	16 ms	16 ms	100 ms
Voltage	0.6 V	0.55 V	0.6 V	0.6 V	0.8 V
DNN Structure	FC	CONV+FC	LSTM+FC	LSTM+FC	TC-ResNet
Bit Width (Weights)	-	7	4/8	4/8	6
Bit Width (Inputs)	-	8	8	8	8
Accuracy[1]	98.35 % (TIDIGITS)[2]	90.51 % (GSCD)	90.87 % (GSCD)	90.00 % (TIMIT)[2,3]	93.09 % (GSCD)
Keywords	11	10	10	4	10
Power	172 µW	52 µW	10.6 µW	5.0 µW	8.2 µW

[1] SNR $\geq$ 1000 dB [2] Note the difference in datasets [3] F1-score

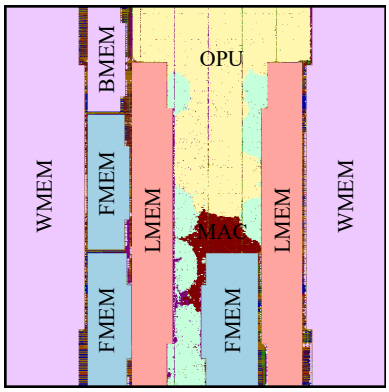

Fig. 15. Die layout of the UltraTrail accelerator in Globalfoundries' 22FDX.

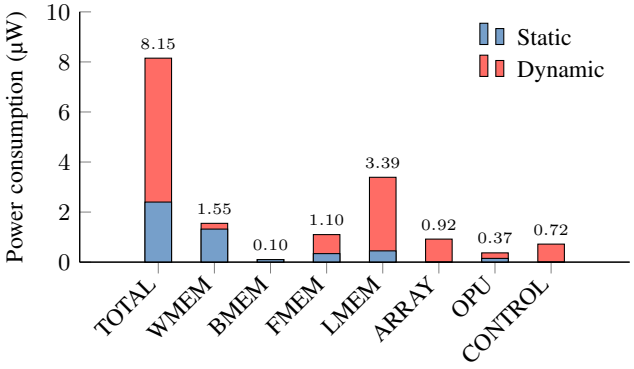

Fig. 16. Power breakdown by components.

without conditional computing, are weighted according to Fig. 13 and are combined to build the final average power consumption. A complete power breakdown for the chip is presented in Fig. 16. The total power consumption is 8.15 µW of which around 75 % are accounted for by the memories. The LMEM stands out from the other components with 42 % of the total power consumption. This is due to the fact that the LMEM is active for the entire duration of an inference as partial sums are continuously accumulated leading to one read and write access by the MAC array in each cycle. In contrast, as a result of the chosen dataflow, the other memories are only active 62 % (FMEM0) / 39 % (FMEM1) / 3 % (FMEM2) / 6 % (BMEM) / 3 % (WMEM) of the time and are clock gated otherwise. The WMEM in particular benefits from the minimized number of accesses and, despite its size, remains at a comparably low power consumption.

The impact of conditional computing on the power consumption is shown in Fig. 17. Compared to an operation where the entire network is always executed, the total power consumption was reduced by 22 % with a 28 % reduction in dynamic power. This reduction is in line with the saved number of clock cycles by taking the early exit which was computed using our runtime analysis of IV-E. A full runtime overview of the network following the structure of Fig. 4 is listed in TABLE III.

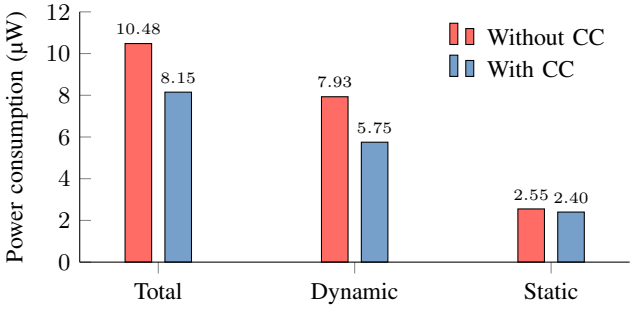

Fig. 17. Impact of conditional computing (CC) on power consumption.

A comparison to other state-of-the-art KWS accelerators is shown in TABLE II. Note, that due to the differences in datasets, technology, and the components contained in the chips, a direct comparison is only possible to a limited extent. Work [13] is mainly designed for automatic speech recognition and differs most from the other chips with a higher clock rate and larger area. Besides the DNN accelerators, [13] as well as [14] and [15] contain further components such as feature extraction or speaker recognition. With 41.3 µW, the CNN accelerator from [14] still has a 5× higher power consumption than this work. The LSTM accelerator used in [15] is based on [21] which has a slightly lower power consumption than this work. Here, as shown in section V-A4, the main advantage

TABLE III
RUNTIME OF OUR TC-RESNET KWS MODEL

	C	C_w	K	F	s	p	$t(l)$ [clock cycles]
CONV-EXT$_0$	40	101	16	3	1	0	2971
CONV-EXT$_{0,0}$	16	99	24	9	2	1	2629
CONV-EXT$_{0,1}$	16	99	24	1	2	0	301
CONV-EXT$_{0,2}$	24	50	24	9	1	1	3871
CONV-EXT$_{1,0}$	24	50	32	9	2	1	2581
CONV-EXT$_{1,1}$	24	50	32	1	2	0	301
CONV-EXT$_{1,2}$	32	25	32	9	1	1	3281
CONV-EXT$_{3,0}$	32	25	12	1	1	0	201
CONV-EXT$_{3,1}$	12	1	12	1	1	0	5
Exit 2	–	–	–	–	–	–	16141
CONV-EXT$_{2,0}$	32	25	48	9	2	1	2521
CONV-EXT$_{2,1}$	32	25	48	1	2	0	313
CONV-EXT$_{2,2}$	48	13	48	9	1	1	3493
CONV-EXT$_4$	48	1	12	1	1	0	13
Normal Exit	–	–	–	–	–	–	22481

of the TC-ResNet lies in its higher accuracy and greater robustness against noise which in our opinion can justify the additional power consumption, depending on the use-case. Overall, this work achieves a notably higher accuracy with a better or comparable power consumption on the GSCD dataset.

VI. CONCLUSION AND FUTURE WORK

This paper presented UltraTrail, a configurable ultra-low power TC-ResNet AI accelerator and its application to efficient keyword spotting. Combining a dataflow architecture that exploits the data dependencies within the temporal convolution with a distributed memory setup which supports the branched nature of modern network architectures we were able to emphasize the potential of TCNs from a hardware perspective. We showed that established DNN accelerator design approaches can be effectively adopted to achieve competitive power results while leveraging the accuracy advantages over CNN or LSTM implementations. The final accelerator attains a KWS accuracy of 93.09 % with a total power of 8.2 µW.

Going forward, we plan to extend the range of applications beyond the current KWS task. Building upon the existing configurability we envision an accelerator suitable for a broad spectrum of sequence modeling and analysis tasks. Based on the results of this work and seeing the continuous advances in neural network design in conjunction with the recent successes of TCNs we believe the field of efficient TCN acceleration to be a promising area for future research.

REFERENCES

[1] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint arXiv:1803.01271*, 2018.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] S. Choi, S. Seo, B. Shin, H. Byun, M. Kersner, B. Kim, D. Kim, and S. Ha, "Temporal Convolution for Real-time Keyword Spotting on Mobile Devices," *CoRR*, vol. abs/1904.03814, 2019. [Online]. Available: http://arxiv.org/abs/1904.03814

[4] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[5] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2016.

[6] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[7] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[8] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[9] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[10] P. Panda, A. Sengupta, and K. Roy, "Conditional deep learning for energy-efficient and enhanced pattern recognition," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 475–480.

[11] W. Hua, Y. Zhou, C. M. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 1884–1894.

[12] K. Goetschalckx, B. Moons, S. Lauwereins, M. Andraud, and M. Verhelst, "Optimized hierarchical cascaded processing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 884–894, 2018.

[13] M. Price, J. Glass, and A. P. Chandrakasan, "A Low-Power Speech Recognizer and Voice Activity Detector Using Deep Neural Networks," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 1, pp. 66–75, Jan. 2018.

[14] B. Liu, Z. Wang, W. Zhu, Y. Sun, Z. Shen, L. Huang, Y. Li, Y. Gong, and W. Ge, "An Ultra-Low Power Always-On Keyword Spotting Accelerator Using Quantized Convolutional Neural Network and Voltage-Domain Analog Switching Network-Based Approximate Computing," *IEEE Access*, vol. 7, pp. 186 456–186 469, 2019.

[15] J. S. P. Giraldo, S. Lauwereins, K. Badami, and M. Verhelst, "Vocell: A 65-nm Speech-Triggered Wake-Up SoC for 10-μW Keyword Spotting and Speaker Verification," *IEEE Journal of Solid-State Circuits*, pp. 1–11, 2020.

[16] Y. Bengio, "Estimating or propagating gradients through stochastic neurons," *CoRR*, vol. abs/1305.2982, 2013. [Online]. Available: http://arxiv.org/abs/1305.2982

[17] P. Yin, J. Lyu, S. Zhang, S. J. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[19] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik, "Neural network distiller: A python package for dnn compression research," October 2019. [Online]. Available: https://arxiv.org/abs/1910.12232

[20] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[21] J. S. Giraldo and M. Verhelst, "Laika: A 5uw programmable lstm accelerator for always-on keyword spotting in 65nm cmos," in *ESSCIRC 2018-IEEE 44th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 2018, pp. 166–169.

Paul Palomero Bernardo was born in Tübingen, Germany, 1996. He received the B.S. and M.S. degrees in computer science from University of Tübingen, Tübingen, Germany, in 2017 and 2020, respectively, where he is currently pursuing the doctoral degree (Ph.D.) at the Department of Computer Science.

His current research interests include neural network hardware and design optimization.

Christoph Gerum was born in Filderstadt, Germany, 1984. He received the Diploma degree (M.S.) in computer science from the University of Tübingen, Tübingen, Germany, in 2011, where he is currently pursuing the doctoral degree (Ph.D.) at the Department of Computer Science.

His current research focuses on performance prediction for embedded systems and hardware-dependent optimization of neural network inference.

Adrian Frischknecht was born in Reutlingen, Germany, 1992. He received the B.S. and M.S. degrees in computer science from University of Tübingen, Tübingen, Germany, in 2015 and 2018, respectively, where he is currently pursuing the doctoral degree (Ph.D.) at the Department of Computer Science.

His current research interests include speech recognition, hardware-software co-design, and energy-efficient neural network accelerators.

Konstantin Lübeck was born in Weimar, Germany in 1989. He received the B.S. and M.S. degrees in computer science from University of Tübingen, Tübingen, Germany, in 2015 and 2018, respectively, where he is currently pursuing the doctoral degree (Ph.D.) at the Department of Computer Science.

His current research interests include performance characterization, modeling, prediction of embedded systems and heterogeneous computer architectures.

Mr. Lübeck received the master's thesis scholarship of the Stiftung Industrieforschung in 2018.

Oliver Bringmann (M'18) received the Diploma degree (M.S.) in computer science from the University of Karlsruhe (KIT), Germany, and the doctoral degree (Ph.D.) in computer science from the University of Tübingen, Germany, in 2001.

He was with the FZI Research Center for Information Technology in Karlsruhe, Germany, in various positions as Department and Division Manager and Member of the management board, until 2012. He has been Professor and Director of the Chair for Embedded Systems at the University of Tübingen since 2012, where he is also serving as Vice Head of the department of computer science since 2014. His current research interests include electronic design automation, embedded system design, timing and power analysis of embedded software, embedded AI architectures, hardware-enhanced security, and robust perception. He is author and co-author of more than 220 publications in the area of electronic design automation, embedded system design, and SoC architectures for automotive electronics and edge devices.