Hardware Accelerator and Neural Network Co-Optimization for Ultra-Low-Power Audio Processing Devices

Gerum Christoph*  
Department of Computer Science  
University of Tübingen  
christoph.gerum@uni-tuebingen.de

Frischknecht Adrian*  
Department of Computer Science  
University of Tübingen  
adrian.frischknecht@uni-tuebingen.de

Hald Tobias  
Department of Computer Science  
University of Tübingen  
tobias.hald@student.uni-tuebingen.de

Paul Palomero Bernardo  
Department of Computer Science  
University of Tübingen  
paul.palomero-bernardo@uni-tuebingen.de

Konstantin Lübeck  
Department of Computer Science  
University of Tübingen  
konstantin.luebeck@uni-tuebingen.de

Bringmann Oliver  
Department of Computer Science  
University of Tübingen  
oliver.bringmann@uni-tuebingen.de

Abstract—The increasing spread of artificial neural networks does not stop at ultralow-power edge devices. However, these very often have high computational demand and require specialized hardware accelerators to ensure the design meets power and performance constraints. The manual optimization of neural networks along with the corresponding hardware accelerators can be very challenging. This paper presents HANNAH (Hardware Accelerator and Neural Network seArcH), a framework for automated and combined hardware/software co-design of deep neural networks and hardware accelerators for resource and power-constrained edge devices. The optimization approach uses an evolution-based search algorithm, a neural network template technique and analytical KPI models for the configurable UltraTrail hardware accelerator template in order to find an optimized neural network and accelerator configuration. We demonstrate that HANNAH can find suitable neural networks with minimized power consumption and high accuracy for different audio classification tasks such as single-class wake word detection, multi-class keyword detection and voice activity detection, which are superior to the related work.

Index Terms—Machine Learning, Neural Networks, AutoML, Neural Architecture Search

I. INTRODUCTION

Artificial intelligence is increasingly spreading into the domain of always-on ultra-low power connected devices like fitness trackers, smart IoT sensors, hearing aids and smart speakers. The limited power budget on these devices and the high computational demand often mandates the use of specialized ultra-low power hardware accelerators, specialized to a specific application or application domain. Hardware design, neural network training and optimized deployment often require manual optimization by the system designers, who need to deal with manifold often counter-directed issues. In this work, we propose HANNAH (Hardware Accelerator and Neural Network seArch) to automatically co-optimize neural network architectures and a corresponding neural network accelerator.

The HANNAH design flow is shown in Figure 1. Neural Networks are instantiated and trained in the training component employing quantization aware training and dataset augmentation. Trained neural networks are then handed over to the deployment component (Sec. III) for target code generation. Here, the neural network is quantized to a low word width representation the neural network operations are scheduled and on-device memory is allocated for the neural network. Along with the target network architecture, a specialized hardware accelerator for the neural network processing is instantiated from a configurable Verilog template. Hardware dependent performance metrics like power consumption and chip area are then either generated by running the neural network on a gate-level simulation or estimated using an analytical performance model. The evolution-based search strategy (Sec. IV) implemented in the HANNAH optimizer is then used to incrementally search the neural network and hardware accelerator codesign space.

The main contributions of this paper are:

1) We present an end-to-end design flow from neural network descriptions down to synthesis and gate-level

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).

*These authors contributed equally to this work.
power estimation.
2) We propose a model-guided hardware and neural network co-optimization and architecture exploration flow for ultra-low power devices.
3) The combined flow reaches state-of-the-art results on a variety audio detection tasks.

II. RELATED WORK

In recent years, neural architecture search (NAS) had an enormous success [1]–[3] in automating the process of designing new neural network architectures. The early work on neural architecture search did not take hardware characteristics into account. More recent works (Hardware-Aware NAS) allow to take the execution latency of a specific target hardware into account and allow to optimize the neural networks for latency as well as accuracy. There are several approaches applying hardware/software co-design to hardware accelerators for neural networks. In early works this involves manually designing specialized neural network operations and a hardware architecture [4]. In [5] and [6] reinforcement learning based NAS is extended to include search for an accelerator configuration on FPGAs and optimize it for latency and area. Zhou et al. [7] for the first time combine differentiable neural architecture search and the search for a neural architecture configuration. All of these approaches are not directly applicable to TinyML, as their search spaces for neural networks and hardware accelerators make it impossible to meet power budgets in the order of <10µW.

NAS for TinyML mostly focuses on searching neural networks for microcontrollers. The search methods in [8], [9] use genetic algorithms and Bayesian optimization to optimize neural networks to fit the constrained compute and memory requirements of these devices. In [10] a weight sharing approach is used to train a super network containing multiple neural networks at once and a genetic algorithm, is used to search for the final network topology. A first approach to apply differentiable neural architecture search to TinyML is presented in [11] it reaches state-of-the-art results on the tinyMLPerf benchmarks, but requires relatively big microcontrollers (~500kB of memory) to execute the found networks. These search methods would generally be applicable to our neural network search problem, but they do not contain a co-optimization strategy for also optimizing the hardware architecture.

For ultra-low power audio processing hardware accelerators or other edge devices with our intended target power there are currently no hardware accelerators. So our main state of the art comprises of manually designed hardware accelerators for a specific neural network architecture. Manual optimization of ultra-low power hardware has been used for all of the use-cases in the experimental evaluation. Recent examples include keyword spotting (KWS) [12]–[16], wake word detection (WWD) [17], [18] and voice activity detection (VAD) [16], [19], [20]. These approaches show impressive results, but all require a work intensive and error-prone manual design process.

III. NPU ARCHITECTURE AND ANALYTICAL MODELING

A. NPU architecture

The NPU architecture used in this work is based on UltraTrail [12], a configurable accelerator designed for TC-ResNet. Fig. 2 shows the basic UltraTrail architecture. The accelerator uses distributed memories to store the features (FMEM0-2), parameters (W/BMEM) and local results (LMEM). Features, parameters and internal results use a fixed-point representation. An array of multiply and accumulate (MAC) units calculates the convolutional and fully-connect layers. A separate output processing unit (OPU) handles post-processing operations like bias addition, application of a ReLU activation function, average pooling. A configuration register stores the structure of a trained neural network layer by layer. Each layer configuration contains, among others, information about the input feature length, input and output feature channels, kernel size, and stride, as well as the use of padding, bias addition, ReLU activation, or average pooling.

The architecture of UltraTrail has been parameterized to suit the executed neural networks as best as possible. At design time, size and number of supported layers of the configuration register, the word width for the features, parameters, and internal results, the size of the memories, the dimension of the quadratic MAC array, and the number of supported classification classes can be modified. During runtime, the programmable configuration register allows the execution of different neural networks.

B. Deployment

To execute a trained neural network with UltraTrail, a schedule, a corresponding configuration, and a sorted binary representation of the quantized weights and bias values have to be created. These steps are part of the automated deployment flow.

The deployment backend converts the PyTorch Lightning model from HANNAH to ONNX and its graph model representation. A graph tracer iterates over the new representation and determines the schedule and the layer-wise configuration which is loaded into the configuration register of UltraTrail. The tracer changes the node order in the obtained graph such that residual connections are processed before the main branch to ensure a correct schedule for the NPU as described in [12]. The graph tracer also searches for exit branches for early exiting to handle correct configuration and treatment when the exit is not taken.
Algorithm 1: Pseudocode of the NPU accelerator when configured with an $N \times N$-MAC array with $K$ output channels, $C$ input channels, $F$ filter size, $S$ stride, $I$ input length, $X = [(I - 1)/S] + 1$ output length, and full padding.

1. for $k = 0 : \lceil K/N \rceil$
2.   for $c = 0 : \lceil C/N \rceil$
3.     for $f = 0 : F$
4.       for $x = 0 : X$
5.         if $i_{idx} \geq 0$ and $i_{idx} < I$
6.           $1^{(N)}[x] += i^{(N)}[c][i_{idx}] \cdot w^{(N \times N)}[k][f][c][f]$
7.     end
8.   end
9. end

The backend extracts weights and biases from the model. These parameters are padded and reordered to fit the expected data layout before they get quantized to the defined fixed-point format. The same happens to example input data provided by HANNAH which is used as simulation input.

The word widths of the internal LMEM memory are calculated by the largest fixed-point format times two plus the logarithm of the maximum input channels to avoid an overflow.

After the schedule and number of weights and biases are fixed, the deployment backend generates the corresponding 22FDX memory macros. The backend determines the size and word width of the weight and bias memory from the fixed numbers. The schedule gets analyzed to determine the size of each FMEM so that the feature maps fit perfectly for the trained neural network. All memory sizes can be adjusted manually to support other neural networks with more parameters or larger feature maps. The LMEM size is determined by the largest supported output feature map. All memory sizes and word widths are set to the next possible memory macro configuration.

Given the generated configuration, weights and biases, memory macros, example input data, and selected hardware parameters, a simulation, synthesis, and power estimation are run automatically if desired. The simulation results are fed back to HANNAH and compared with the reference output to validate the functional correctness of the accelerator.

C. NPU Models

The simulation and synthesis of UltraTrail are time-consuming and therefore not feasible for hundreds to thousands of possible configurations. To get an accurate yet fast estimation of UltraTrail, performance, area, and power consumption are estimated using analytical models.

For latency estimation we adopt the cycle-accurate analytical model presented in [12] to configurable array sizes. The pseudocode shown in Algorithm 1 visualizes the NPU operations and memory accesses. The accelerator iterates over $C$ input and $K$ output channels tiled by the MAC array size $N$. One $N \times N$ patch of the weight array is fetched from the weight memory per iteration of the next loop. In the innermost loop $N$ input channels are fetched from one of the FMEMs and the current convolution outputs are accumulated in the LMEM using a spatially unrolled matrix-vector multiplication on the MAC array. To avoid misaligned memory accesses and accelerate the computation, padding is implemented by skipping the corresponding loop iterations instead of actual zero padding of the feature maps. In the last loop, the OPU fetches $N$ output channels from the LMEM and the $N$ results are stored in a planned FMEM. The latency of the accelerator without skipping of padded values can be easily estimated using the following equation, by just counting the number of loop iterations.

$$l = 1 + \left\lceil \frac{C}{N} \right\rceil \cdot \left\lceil \frac{K}{N} \right\rceil \cdot F \cdot X$$ (1)

Where the output length $X$ can be derived from input length $I$ using $X = [(I - 1)/S] + 1$. The crucial part of the performance model is to accurately calculate the number of skipped loops in Line 7 of Algorithm 1. For the case $i_{idx} \geq 0$ skipping happens in the first $a_{p,b} = \left\lceil \frac{F/2 - 1}{S} + 1 \right\rceil$ iterations over $x$, the number of skipped executions is $\left\lfloor \frac{F}{2} \right\rfloor$ at the start of the loop, and decreases by $S \cdot i$ for each iteration over $X$. For the case $i_{idx} < I$ the analytical model is similar but special care must be taken if the input length $I$ is not divisible by the stride. In the last iteration over $x$, $i_{idx}$ takes the value $I_{max} = (X - 1) \cdot S - \left\lfloor F/2 \right\rfloor + F - 1$ leading to an effective padding of $C_{w,e} = I_{max} + 1 - \left\lfloor F/2 \right\rfloor - I$. Loop skipping then happens for the last $a_{p,e} = \left\lceil \frac{C_{w,e} - 1}{S} + 1 \right\rceil$ iterations. The number of skipped iterations is $C_{w,e}$ at the end of the loop and also decreases by $s \cdot i$ at each loop iteration. This leads to an estimation of the total number of skipped MAC array operations at the beginning $\#MAC_{not,b}$ and end $\#MAC_{not,e}$ of the loop as:

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

$$\#MAC_{not,e} = \sum_{i=0}^{a_{p,e} - 1} \left\lfloor C_{w,e} \right\rfloor - s \cdot i$$ (3)

And a total per layer latency of:

$$l = 1 + \left\lceil \frac{C}{N} \right\rceil \cdot \left\lceil \frac{K}{N} \right\rceil \cdot F \cdot X - \#MAC_{not,e} - \#MAC_{not,b}$$ (4)

The power model has two parts for SRAM power estimation and for other NPU components. For every layer and memory, the model calculates the number of read, write, and nop (idle) operations. LMEM and input feature memory (IMEM) are accessed at each cycle of the operation except for a single setup cycle. The read cycles $r_{mem}$ and write cycles $w_{mem}$ are the same as the layer latency $l - 1$. As the number of memory accesses to the weights remain stationary during the
innermost loop of Algorithm 1, the weight memory is accessed only \( \lceil \frac{C}{N} \rceil \cdot \lceil \frac{K}{N} \rceil \cdot F \) times \( \tau_{\text{weight}} \). Memory access to the output feature memory \( \tau_{\text{outmem}} \), the bias memory \( \tau_{\text{bmem}} \), and in the case of residual blocks the partial sum feature memory \( \tau_{\text{psmem}} \) is given as \( \lceil \frac{K}{N} \rceil \cdot X \). Idle times for the memories are then calculated using the layer latencies and access times for each memory: \( i_m = l - r_m - w_m \).

Furthermore, the combinational MAC array alone causes many relevant glitches related to SRAM. Therefore, the glitching LMEM data input leads to a non-negligible power increase. Based on some gate-level simulations for different networks and array sizes, a linear regression for the number of glitches \( g_{\text{mem}} \) is used.

Finally, if the network latency on the accelerator \( L \) is below the period \( P \) for real time operation, the memories are switched to low power modes with reduced leakage for the remainder of the period. The final power consumption of the memories is then estimated using:

\[
P_{\text{mem}} = \frac{L}{P} \cdot P_{\text{inmem}} \cdot g_{\text{mem}} + \sum_{m} \frac{L}{P} (P_{\text{read}} \cdot r_m + P_{\text{write}} \cdot w_m + P_{\text{idle}} \cdot i_m)
\]

\[
+ \frac{L}{P} \cdot P_{\text{static}} + \left( 1 - \frac{L}{P} \right) \cdot P_{\text{lp}}
\]

The dynamic power \( P_{\text{read}} \), \( P_{\text{write}} \), \( P_{\text{idle}} \) and static power in running- \( P_{\text{static}} \) and low power-mode \( P_{\text{lp}} \) are extracted from the memory compiler for each memory macro usable during the co-optimization and stored in a database.

Again, the model assumes a constant value for the control unit as the power consumption is mainly independent of the configuration and executed neural network. The power consumption of the MAC array and OPU is approximated using a linear regression on MAC array size and word width of the MAC array calibrated using gate-level simulations. Note, that the dynamic power for non-memory modules is weighted depending on the runtime per inference.

The analytical area model comprises an exact SRAM area calculation and an estimation for the other NPU components to estimate the cell area of the NPU after synthesis. The model looks up the SRAM area in a small database containing all necessary memory macros used during NAS. The SRAM area estimation is by far the most important as it is responsible for about 90% of the total synthesis cell area. The control unit is mainly independent of the configuration and its area is modelled as constant. The MAC array is estimated by a linear growth of the word width and quadratic growth of the MAC array dimension based on a minimum MAC unit area. The OPU model is like the MAC array model however grows linear with MAC array size.

IV. NEURAL NETWORK / HARDWARE CO-OPTIMIZATION

The co-optimization uses a block based search space for neural network architectures. As shown in Figure 3a each block is either a residual block with a main branch of a configurable number of convolutional layers on the trunk and a skip connection or a simple feed forward CNN block leaving out the skip connection. We additionally search over layer, block and network level hyperparameters like filter size, stride and convolution sizes. The hardware accelerator search space consists of the MAC array dimensions, the memory sizes and multiplier word widths. The mac array size is optimized using the search algorithm while the other metrics are derived from the neural network parameters, during neural network deployment. A full overview of the parameters is shown in Table I.

The search for a target neural network and accelerator architecture configuration is implemented as an evolution-based multi-objective optimization. As shown in Algorithm 2, the search first samples random neural network and hardware configuration from a joint search space \( S \). The neural network is then trained using quantization aware training, and the trained neural network is then evaluated on the validation set to obtain the accuracy metric. All further performance metrics are estimated by the analytical hardware model as described in Section III. At the end of an optimization step, the
sampled architecture parameters and the performance metrics are added to the search history. After the initial population size $s$ has been reached, new architectures are derived from the current population using element-wise mutations. During search, HANNAH randomly samples from the following set of mutations:

1) Add/remove a block
2) Change block type between residual and feed-forward
3) Add/remove a convolutional layer
4) Increase/decrease convolution size
5) Increase/decrease major block stride
6) Increase/decrease quantization word width
7) Increase/decrease MAC array size
8) Increase/decrease number of output channels

To select the ancestor of the next evaluation point. Similar to current neural network search for TinyML on microcontrollers [8], [9], we adopt randomized scalarization [21]. In this approach the architecture parameters are ranked according to the following scalarization function:

$$ f(M) = \max_{m_i \in M} \lambda_i \cdot m_i \quad (6) $$

The $\lambda_i$ are sampled from the uniform distribution over $[0, \frac{1}{b}]$, where $b_i$ denotes a soft boundary for each target metric. Choosing $\lambda_i$ in this way ensures that candidates violating soft targets are always ranked behind targets that satisfy a target, while on the other hand encouraging the search to explore different parts of the search space near the Pareto boundary.

V. EXPERIMENTAL EVALUATION

The HANNAH framework has been implemented using current best practice libraries, using PyTorch version 1.10.1 [22]. Training hyperparameters have been set to fixed values for all experiments. All training runs use 30 epochs and use a batch size of 128 samples per minibatch. The optimizer used for training the network parameters is adamW with a one-cycle learning rate scheduling policy using a maximum learning rate of 0.005. All other optimizer and learning rate parameters are left at the default values provided by PyTorch. All searches are run on a machine learning cluster using 4 Nvidia RTX2080 Ti GPUs. We train 8 neural network candidates in parallel which takes approximately 10 minutes. The training uses noisy quantization aware training [23]. The training sets are augmented using the provided background noise files for keyword spotting and voice activity detection and using random white noise for wake word detection. For inference the batch norm weights are folded into the weights and bias of the preceding convolutional layer [24].

As the hardware accelerator is set to operate at 250 kHz and the neural networks are trained and evaluated with an input time shift of 100 ms we set the latency constraint of the neural network architectures to 25,000 cycles corresponding to the maximum input shift used during training and evaluation. The area constraints are set to 150,000 µm$^2$ which is slightly less than the configuration used in [12] for KWS. The constraints for KWS are set to 5 µW maximum power and accuracy constraints are set to 93.0%. The other tasks use a power budget of 1 µW and accuracy constraints of 95.0%. All searches ran with a search budget of 3000 individual architectures and use a population size of 100.

We use the 22FDX technology by GlobalFoundries for implementation with low-leakage standard cells and SRAMs from Invecas. For synthesis and power estimation we use Cadence Genus 20.10 and Cadence Joules 21.11, respectively. For the average power consumption, we use two separate power estimations for the inference with previous feature loading and idle time. A weighted sum adds these two parts accordingly. The load of weights and bias is not included as it must be performed only once. It is evaluated at a 25 °C TT corner with 0.8 V supply voltage and no body bias voltage. The NPU uses clock-gating and low-power modes of the SRAM during idle times and waiting for the next inference to start.

A. Results of Neural Network Co-Optimization