Special Session: Estimation and Optimization of DNNs for Embedded Platforms

Axel Jantsch,
Christian Doppler Laboratory for Embedded Machine Learning
TU Wien, Vienna, Austria
Email: axel.jantsch@tuwien.ac.at

Song Han,
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
Email: songhan@mit.edu

Lin Meng,
Department of ECE
Ritsumeikan University
Kusatsu, Shiga, Japan
Email: menglin@fc.ritsumei.ac.jp

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

Haotian Tang
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
Email: kentang@mit.edu

Shang Yang
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
Email: shangy@mit.edu

Hengyi Li,
Department of ECE
Ritsumeikan University
Kusatsu, Shiga, Japan
Email: lihengyi321@gmail.com

Matthias Wess
Christian Doppler Laboratory for Embedded Machine Learning
TU Wien, Vienna, Austria
Email: matthias.wess@tuwien.ac.at

Martin Lechner
Christian Doppler Laboratory for Embedded Machine Learning
TU Wien, Vienna, Austria
Email: martin.lechner@tuwien.ac.at

***Abstract*—Several state of the art estimation and optimization techniques for CNNs and LLMs on embedded devices are summarized. For LLMs an Activation-aware Weight Quantization and on-the-fly dequantization techniques is presented. For CNNs various pruning algorithms and an integrated optimization and implementation flow is discussed. To estimate inference latency of CNNs on specific hardware platforms, three different techniques are reviewed: A mixed analytic-stochastic model, an analytic model based on step-wise linear functions, and a method that uses a detailed architecture description of the hardware.**

***Index Terms*—Embbeded Machine Learning, CNN, LLM, Accelerator, Estimation**

I. Introduction

Efficient implementation of a Deep Neural Network (DNN) on a given platform under tight constraints is challenging due to many non-linear dependencies. Small changes in the DNN or the platform configuration often have disproportional effects on the performance.

In this paper, we explore state-of-the-art methods for optimization (sections II and III) and estimation (sections IV and V) of DNNs on tightly constrained embedded platforms.

Section II presents optimization techniques for implementing state-of-the-art visual language models on embedded platforms. By using Activation-aware Weight quantization, that selectively quantizes only less important weights, and an on-the-fly dequantization technique at inference time, the memory footprint and the inference performance of Large Language Models (LLMs) is greatly improved, thus facilitating their deployment in embedded devices such as Jetson Orin. Then, section III reviews optimization and implementation methods to customize Convolutional Neural Networks (CNN) for tightly constrained embedded devices. The section focuses on pruning and studies channel-level pruning, layer-level pruning and a hardware-aware pruning technique. Further, the section describes an integrated hardware aware optimization and implementation flow for CPU and FPGA based platforms. In addition, a CNN based object detection application for embedded robots is elaborated.

Fast and accurate estimation facilitates the efficient exploration of the design space without the need to implement many model variants on various hardware platform, potentially saving orders of magnitude of design time. Section IV discusses two state of the art estimation tools, ANNETTE and Blackthorn, for latency estimation of models on various platforms. ANNETTE uses a mixed analytic-stochastic models to estimate the latency of individual or fused layers on CPU-, FPGA-, and NPU-based hardware, while Blackthorn uses step-wise linear analytic functions for latency estimations on GPUs and NPUs. Finally, section V presents performance estimation based on an abstract computer architecture description language to capture relevant features of the target platform. Given architecture model of the AI accelerator and the DNN dataflow, the usage of buffers, memories and compute units are tracked in detail and thus facilitate accurate latency estimations.

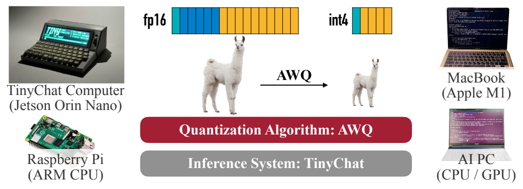

fp16
int4
AWQ
TinyChat Computer (Jetson Orin Nano)
Raspberry Pi (ARM CPU)
Quantization Algorithm: AWQ
Inference System: TinyChat
MacBook (Apple M1)
AI PC (CPU / GPU)

Fig. 1. We introduce **AWQ**, a versatile weight quantization method for LLM. To implement AWQ, we developed **TinyChat** to deploy 4-bit quantized LLMs into various edge platforms, achieving a **3-4**× performance boost compared to FP16. Notably, we've also manufactured a **TinyChat computer**, powered by TinyChat, which contains an NVIDIA Jetson Orin. Demo: https://youtu.be/z91a8DrfgEw.

II. LLM Quantization for Edge Deployment

Large Language Models (LLMs) have revolutionized AI applications, but their deployment on edge devices remains challenging due to their enormous size and computational requirements. To solve these issues, we propose Activation-aware Weight Quantization (AWQ) and TinyChat, an algorithm-system full-stack solution for efficient on-device LLM deployment. AWQ is a novel quantization method that identifies and protects salient weights based on activation distribution, significantly reducing model size while preserving performance. TinyChat, an optimized inference framework, translates AWQ's theoretical memory savings into practical speedups through techniques like on-the-fly dequantization, SIMD-aware weight packing, and kernel fusion. Together, they enable **4**× model size reduction and **3-4**× acceleration across various edge platforms, from high-end desktop GPUs to resource-constrained IoT devices. This solution democratizes on-device LLM deployment, offering privacy-preserving, low-latency AI capabilities across a wide range of applications.

A. AWQ: LLM Low-bit Weight Quantization

Deploying large language models (LLMs) on edge devices is crucial. It eliminates delays caused by sending data to a cloud server and better protects user's privacy. However, the large model size leads to the high serving costs. For example, GPT-3 has 175B parameters, which is 350GB in FP16, while the latest B200 GPU only has 192GB memory, let alone edge devices. Quantization for LLMs can significantly reduce the memory footprint of LLM inference.

Protecting Salient Channels. In this work, we first propose Activation-aware Weight Quantization (AWQ), a hardware-friendly low-bit weight-only quantization method for LLMs illustrated in Figure 2. The first key observation in AWQ is that the weights of LLMs are *not equally important*: there is a small fraction (0.1%-1%) of *salient* weights that are much more important for LLMs' performance compared to others. Skipping the quantization of these salient weights can help bridge the performance degradation due to the quantization loss *without* any training or regression.

Activation Awareness. However, identifying salient weight channels can be challenging. Contrary to conventional wisdom, we find that the importance of weights should be determined by the *activation* distribution rather than the *weight* distribution itself, despite we are doing *weight-only* quantization: weight channels corresponding to larger activation magnitudes are more salient since they process more important features. Keeping those weights in FP16 can preserve the salient features, which contributes to better model performance.

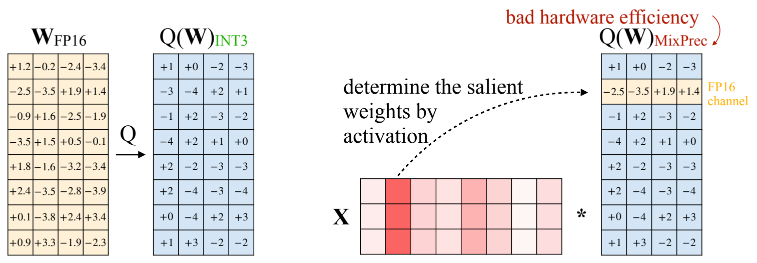
To leverage this insight without resorting to inefficient mixed-precision implementations, AWQ employs a mathematically equivalent scaling method. The algorithm scales up the salient weight channels before quantization, effectively reducing their relative quantization error. Intuitively, scaling up a channel by 2× effectively *adds one more bit*, assuming other channels are unchanged. For example, both 0.52 and 1.1 round to 1; but 1.04 and 2.2 will round differently and can be distinguished. This scaling is determined by collecting activation statistics offline, making the process efficient and generalizable across different domains and modalities.

One of the key advantages of AWQ is its simplicity and efficiency. Unlike methods that rely on back-propagation or complex reconstruction processes, AWQ does not require fine-tuning or extensive calibration data. This makes it particularly well-suited for quantizing large pre-trained models, including instruction-tuned LMs and multi-modal LMs.

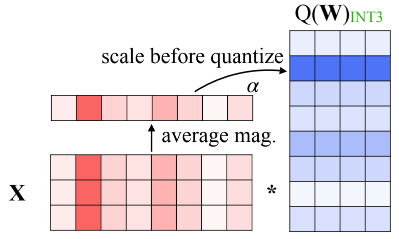
Results. As in Figure 3, AWQ's effectiveness has been demonstrated across a wide range of model sizes and architectures. For example, when applied to the LLaMA [1], [2] model families, AWQ consistently outperforms both round-to-nearest (RTN) quantization and more complex methods like GPTQ [3] and GPTQ-R (GPTQ with reordering). For the Llama-2 70B model with INT3-g128 quantization, AWQ achieves a perplexity of 3.74, compared to 3.98 for RTN and 3.86 for GPTQ-R. Moreover, AWQ shows remarkable generalization capabilities. It performs well not only on standard language modeling tasks but also on multi-image visual language models such as VILA [4]. AWQ achieves lossless quantization performance on 11 visual-language benchmarks evaluated in the VILA paper. To sum up, our method provides a push-the-button solution for LLM/VLM quantization. It is the *first* study of VLM low-bit quantization to the best of our knowledge.

B. TinyChat: Efficient Inference Engine for 4-bit LLMs

While AWQ substantially reduces the size of LLMs, converting the theoretical memory savings from W4A16 (4-bit weight, 16-bit activation) quantization into measured speedup is non-trivial. Alternative W8A8 quantization methods, such as SmoothQuant [5], maintain *the same* data precision for both storage and computation. This allows the dequantization procedure to be seamlessly integrated into the computation kernel's epilogue. On the other hand, W4A16 quantization employs *different* data types for memory access and computation. As a result, its dequantization must be incorporated into the primary computation loop for optimal performance, posing implementation challenges. To tackle this, we introduce TinyChat, a nimble system for AWQ model inference. It boasts a PyTorch frontend and a backend harnessing device-specific instruction sets (e.g., CUDA/PTX, Neon, AVX).



(a) RTN quantization (**PPL 43.2**) (b) Keep 1% salient weights in FP16 (**PPL 13.0**)



(c) Scale the weights before quantization (**PPL 13.0**)

Fig. 2. We observe that we can find 1% of the salient weights in LLMs based on the *activation distribution* (middle). Keeping the salient weights in FP16 can significantly improve the quantized performance (PPL from 43.2 (left) to 13.0 (middle)), but the mixed-precision format is not hardware-efficient. We follow the activation-awareness principle and propose AWQ (right). AWQ performs per-channel scaling to protect the salient weights and reduce quantization error. We measure the perplexity of OPT-6.7B under INT3-g128 quantization.

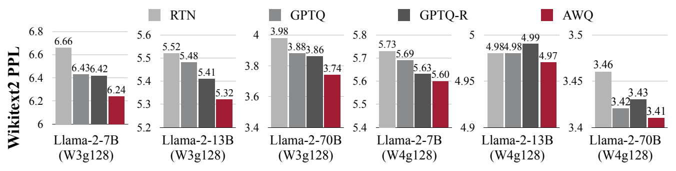


Fig. 3. AWQ improves over round-to-nearest quantization (RTN) for different model sizes and different bit-precisions. It consistently achieves better perplexity than GPTQ (w/ and w/o reordering) on Llama-2 models.

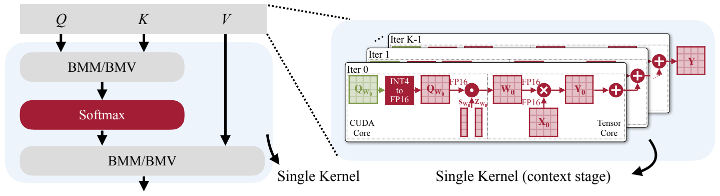
To understand the acceleration opportunities for quantized LLMs on edge devices, we profiled the LLaMA-7B model on an RTX 4090 GPU. Our analysis revealed three key insights: (1) The generation stage is substantially slower than the context stage, particularly for on-device interactive applications. (2) The generation stage is memory-bound, with an arithmetic intensity of approximately 1 for single-batch FP16 execution. (3) Weight access dominates memory traffic. The amount of weight access is orders of magnitude larger than the amount of activation access. These findings suggest that weight-only quantization to 4-bit integers can theoretically increase the arithmetic intensity to 4 FLOPs/Byte, potentially leading to a 4× improvement in peak performance. This analysis provides the foundation for TinyChat's design, featured by the following optimizations.

On-the-fly weight dequantization For quantized layers, as the hardware does not provide multiplication instructions between INT4 and FP16, we need to dequantize the integers to FP16 before performing matrix computation. We avoid writing dequantized weights into DRAM by fusing dequantization kernels with the matrix multplication kernel. Note that we designed a unified weight storage format for context stage (which employs GEMM) and decoding stage (which employs GEMV) linear kernels. We illustrate the context stage dequantization kernel fusion implementation in Figure 4(a), where we convert INT4 weights to FP16 in the register file before sending the converted half-precision weights to tensor cores for computation.

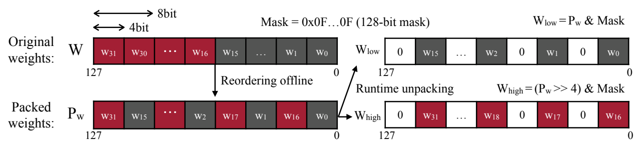
Kernel fusion. We also extensively apply kernel fusion for other layers. For layer normalization, we fuse all operators (e.g., multiplication, division and square root) into a single kernel. For attention layers, we fuse QKV projections into a single kernel, and also perform on-the-fly positional embedding calculation. We also pre-allocate KV caches and perform cache updates within the attention kernel. Kernel fusion is particularly useful for models with inefficient forward pass implementations, such as Falcon [6] and StarCoder [7]. Notably, the computation time for each FP16 kernel is in the order of 0.01ms on the 4090 GPU, comparable to the GPU kernel launch overhead. Hence, reducing number of kernel calls through kernel fusion leads to direct speedups.

SIMD-aware weight packing. On-the-fly weight dequantization reduces intermediate DRAM access, but remains expensive. For instance, dequantizing *a single 4-bit weight* involves 1 shift, 1 bitwise AND, and 1 FMA scaling operations, while the dequantized weight undergoes only 1 FMA computation. This process is particularly costly on CPUs with SIMD architecture that favor vectorized instructions. To mitigate this, we suggest platform-specific weight packing tailored to the bitwidth of a device's SIMD units. Fig. 4(b) demonstrates our strategy for ARM CPUs with 128-bit SIMD registers offering up to 1.2× speedup. Here, each register holds 32 4-bit weights, sequenced as $w_0, w_{16}, w_1, w_{17}, ..., w_{15}, w_{31}$. This approach requires just three SIMD instructions to unpack *all 32 weights*, as opposed to 3 scalar instructions *per weight* in a conventional packing $(w_0, w_1, ..., w_{31})$. Generally, for 2^n-bit SIMD registers, adjacent weights will have indices off by $1/8 \times 2^n$, since each register can hold $1/8 \times 2^n$ 8-bit integers. On GPUs, we found it more efficient to pack each 8 weights into $w_{\{0,2,4,6,1,3,5,7\}}$ following [8].

Results. As in Figure 5, TinyChat achieves significant speedups (**2.7-3.9**×) for various LLM families (Llama-2, MPT, Falcon) on NVIDIA GPUs. For Llama-2-7B on a 4090 GPU, it improves inference speed from 52 to 62 tokens/s through FP16 kernel fusion, with an additional **3.1**× speedup from quantized linear kernels. On the laptop 4070 GPU with only 8GB memory, it runs Llama-2-13B at 33 tokens/s where FP16 implementations can't fit 7B models. TinyChat even enables 7B model deployment on resource-constrained devices like Raspberry Pi 4B at 0.7 tokens/s. The system also accelerates visual-language models like VILA-7B and VILA-13B by **3**× on NVIDIA Jetson Orin.



(a) **Kernel fusion** for attention and GEMM layers on GPU platforms



(a) **SIMD-aware Weight Packing** for ARM CPUs

Fig. 4. (a) TinyChat extensively performs **kernel fusion** for both GEMM and attention layers, reducing intermediate DRAM access and saving kernel calls. (b) TinyChat saves weight dequantization overhead through **SIMD-aware Weight Packing** on ARM CPU platforms.

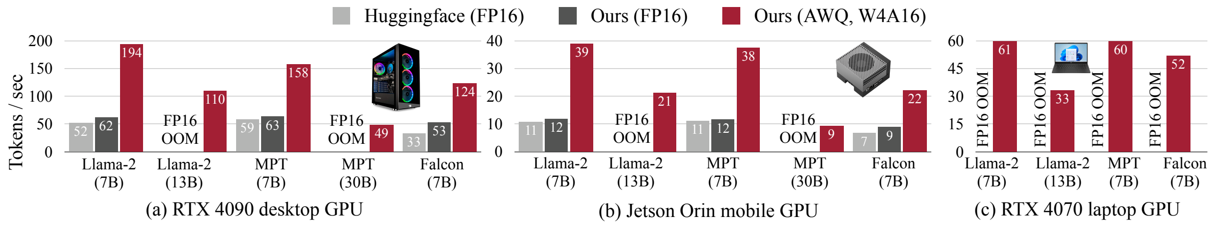


Fig. 5. TinyChat provides a turn-key solution to transform the theoretical memory footprint reduction into a quantifiable speedup. As a result, TinyChat is up to **3.9**× and **3.5**× faster than the FP16 implementation from Huggingface on RTX 4090 (desktop GPU) and Jetson Orin (mobile GPU), respectively. AWQ also democratizes Llama-2-13B deployment on laptop GPUs (4070) with merely 8GB memory.

III. DNN Model Optimization and Implementation for Embedded Systems

Deep neural networks' (DNNs') heavy workload and inefficient computations have always been major obstacles, making widespread use and application difficult. To tackle these challenges, we've conducted deep learning analyses focused on speeding up DNN inference, specifically in hardware-constrained scenarios like embedded systems. Drawing from these insights, our presentation showcases compression techniques tailored for deploying DNNs on embedded devices, including innovative layer-wise pruning. The compressed DNN models have been implemented on FPGA, GAP8, and Raspberry Pi, with experimental results showing significant improvement in inference time. Additionally, the presentation features an application that combines DNN and robotics for dish recycling automation, demonstrating substantial progress in optimizing object detection models for limited hardware.

A. Software-level Optimization

The current typical DNN architectures exhibit substantial inherent redundancies. Specifically, the layer-wise running time studies of these DNN architectures reveal that convolution operations account for approximately 42.30% to 90.34% of the total inference time on single instruction multiple data (SIMD) CPUs. And the input sparsity in convolutional layers reaches an average of 69.61% [9]. These findings suggest that the current DNN architectures could be significantly optimized by reducing redundant operations.

The software-level solutions for optimizing DNNs could be simply categorized into four groups: pruning and quantization, low-rank factorization, transferred/compact convolutional filters, and knowledge distillation [10], [11]. Among these, pruning stands out as the most efficient solution because it fundamentally simplifies the network architecture by removing redundant structures. This approach not only alleviates the overfitting effects of DNNs to a certain extent [12] but also integrates seamlessly with the other four optimization techniques. Given the notable advantages of pruning, our research primarily focuses on the development and improvement of pruning techniques. Specifically, we put forward four structured pruning methods for compressing DNNs: the refined channel-level pruning method SI-Pruning, layer-level pruning mechanism LL-Pruning, a hardware-aware approach IHSOpti, and a two-stage generic optimization method GC&AO.

The key challenge in pruning lies in identifying the redundant structures within the network. SI-Pruning addresses this by utilizing two key factors: the input sparsity of the convolutional layers, which quantitatively indicates network redundancies, and the weights of batch normalization (BN) layers, which denote the importance of layer-wise channels. These factors help identify the superfluous channels in the network. The identified redundant channels are then iteratively pruned, effectively compressing the model.

SI-Pruning as well as traditional methods, are generally effective for channel-level pruning but not for layer-level optimization. To address the shortcoming, we further propose the LL-Pruning, which employs the stochastic optimization algorithm HSPG [13] and a residual strategy [14] to identify not only channel-level redundancies but also layer-level redundancies in DNNs.

IHSOpti aims to fully harness the potential of modern hardware parallelism, with a particular emphasis on pipelining mechanisms [15]. Specifically, IHSOpti has developed an advanced sparse training algorithm, Polar_HSPG, which incorporates a newly proposed layer-wise refined polarization regularizer (LWPolar), based on the half-space project gradient (HSPG) [16]. Furthermore, IHSOpti introduces an innovative residual strategy to optimize layer-level redundancies in neural networks, leveraging the inherent pipelining attributes of hardware. Experimental results demonstrate that IHSOpti achieves exceptional pruning ratios in both parameters and FLOPs.

The frequent data movement required between the processor and off-chip memory limits system performance and energy efficiency [17]. This problem is exacerbated on edge devices with constrained memory bandwidth. Data movement operations stem from the multi-layer structure of DNNs, involving tasks such as rearrangement (Im2col) [18], re-quantization [19], and storage. This fact encourages us to propose a generic deep learning architecture optimization method to achieve further acceleration on edge devices. The CNNs are optimized in two stages: Global Constraint (GC) and Architecture Optimization (AO) [20]. The GC stage aims to achieve lossless channel pruning. The main paths are constrained by the adjunct layers. While AO phase aims to identify residual blocks that have lost function due to constraints and remove them.

B. Implementation for Embedded Systems

At the hardware level, we aim to implement and accelerate software-optimized DNNs on various hardware platforms, including SIMD CPUs, FPGAs, and particularly RISC-V devices, which offer a highly promising solution for embedded DNN applications.

The acceleration of DNNs at the SIMD-instruction level is realized by eliminating the meaningless calculations concerning the sparsity elements at the SIMD level [21]. The proposal achieves up to 28.02% improvement for certain layers of VGG architecture. There are also significant limitations for the proposal due to the reason that the method comes up with too many branches and branch misses, which seriously break the pipelining of CPUs.

The FPGA-based solution combines both hardware-level and software-level optimization for DNNs. Considering the great flexibility of hardware reconfigurable, the nature of high-efficient parallel computing, power efficiency, etc., FPGA generally provides a promising solution for hardware-based DNN accelerations. The implementation of DNNs on FPGA utilizes the development stack Vitis AI which is for AI inference on xilinx hardware platform. First, the optimized VGG13BN and ResNet101 utilizing the proposed SI-Pruning method are deployed on FPGA device ZCU102. The experimental results show that the acceleration achieves up to 151.99 and 124.31 frames per second (FPS) for the two networks, respectively. Second, the results of the GC method and GC&AO method demonstrate significant latency reduction by the AO stage. For CIFAR10, 21.90% latency is reduced in GC stage, while the GC&AO method further reduces latency by 21.40%. On the CIFAR100, it is also observed that the GC&AO method further reduces latency by 24.00% compared to the GC method. In addition, the accuracy achieved in GC and GC+AO methods is similar. These data demonstrate that the proposal achieves further acceleration on the pruned model.

Moreover, the optimized networks have been deployed on the advanced RISC-V platform, specifically the GAP8 SoC. The GAP8 is a highly efficient ultra-low-power parallel computing platform specifically designed for embedded applications [22]. It integrates eight RISC-V cores and a neural network acceleration engine, enabling it to execute deep learning algorithms in resource-constrained environments effectively. However, the limited memory capacity of the GAP8, particularly its L2 memory, imposes significant constraints on the deployment and execution of algorithms. The IHSOpti method presented in this article offers an effective solution to the aforementioned deployment challenge. And the DNN networks optimized by IHSOpti are further accelerated on GAP8 utilizing the GAP8*flow* tool in the research.

GAP8*flow* is the official deployment tool-chain for GAP8, comprising two primary components: NNTool and AutoTiler [22]. NNTool performs static topology optimization, such as node fusion, to enhance operational efficiency and minimize redundant computations. Additionally, NNTool supports quantization using calibrated datasets to convert floating-point models to low-bit-width integer models (e.g., 8-bit). This conversion substantially reduces computational and storage resource consumption while preserving the model's high-precision inference capabilities. Finally, NNTool outputs the model for AutoTiler. The Autotiler tool is designed to optimize data movement and computational efficiency by optimizing data transfer within the memory hierarchy and calculating optimal tiling sizes. Furthermore, it generates GAP code employing double- or triple-buffered mechanisms, which leverage optimized software library primitives to boost computational parallelism and data processing speed.

VGG16BN architecture is adopted in the experiment for its superior representational capability performance in feature extraction. Before deployed on GAP8, the network is optimized with the IHSOpti algorithm, resulting in a parameter reduction of 98.82% and a pruning rate of 91.28% [23]. Compared to the original model, which can not be deployed due to excessive memory consumption, the IHSOpti-optimized model uses only 54% of the L2 memory on the GAP8 platform, achieving an inference time of only 115.26 milliseconds.

C. Object Detection DNN Model Optimization for Embedded Robot

In addition to the aforementioned sections focusing on classification-task DNN architectures, we have developed multiple optimized object detection DNN models to address the vast size and complex computations posed by conventional object detection models. Initially, the YOLO-GS model is proposed, featuring a modified CSPDarknet backbone structure [24]. YOLO-GS incorporates an ultralightweight neck structure for efficient feature fusion, a lightweight head structure for object classification, and bounding box coordinate regression utilizing ghost shuffle convolution (GSConv2D) and the anchor-free method. The model is further optimized and accelerated with TensorRT for efficient and intelligent dish detection on the NVIDIA Jetson Xavier NX. Experimental results demonstrate that YOLO-GS achieves an mean average precision (mAP) of 99.38% with a parameter count of 0.61 M. The inference speed reaches 31.371 FPS, meeting the needs of real-time dish detection on Embedded Robot. A demo image is available at https://www.youtube.com/watch?v=pCBo1nzm3qU&t=22s.

Furthermore, we have designed a multiscale stereoscopic attention (MSA) network called YOLO-MSA to detect post-prandial dishes for empty-dish recycling robots [25]. First, the standard convolution is replaced with a Res2Net module to enhance the network's multiscale expressiveness at a finer-grained level. Second, the MSA module is designed for coarse-grained multiscale expression, employing Res2Net modules with different dilation rates and a novel stereoscopic attention mechanism. Third, to facilitate multiscale feature learning during dimensionality reduction, the Dimension Reduction Spatial Pyramid Pooling (DRSPP) method is proposed to fuse feature maps from different scales. Extensive experiments show that YOLO-MSA has achieved an mAP of 98.47% with an inference speed of 33.93 FPS. The results on other public datasets also demonstrate and provide the proposed model with better generalization capability.

Additionally, to tackle the challenges posed by precise dataset creation and the intricate construction of deep learning models, we have developed an Internet of Things (IoT)-based dataset creation method and an automatic deep learning model generation strategy [26]. Specifically, our IoT-based dataset creation approach facilitates convenient data collection and accurate dataset construction. It involves an Android application that captures images and transmits them to a cloud server for dataset creation. This process includes a two-level verification mechanism to filter out anomalous data and ensure dataset integrity. Subsequently, leveraging the curated dataset, the cloud server employs state-of-the-art deep learning components to automatically train and generate a deep learning model. This methodology has been successfully applied to Empty-dish Recycling Robots, achieving an impressive accuracy of 99.86% with a compact parameter size of just 0.84MB.

IV. Latency estimation

Even when a small set of fixed Deep Neural Networks (DNNs) is given, the design space of DNN based applications in embedded devices is vast, complex and full of peculiar interdependencies (figure 6). The embedded processors are composed of discrete resources like buffers, caches and processing elements. If a slightly larger network exceeds a threshold and a given memory or resource array is too small to serve all of it, we observe stalling and flushing effects that result in highly non-linear performance measurements [27]. Also, a given HW treats different algorithms differently reflecting the objectives of their designers. E.g. depth-wise convolutions consume more energy per operation than ordinary convolutions for Intel's NCU and Google's TPU [28], 4-6 times more on average and up to 20 times in specific cases. Because the efficiency of execution varies greatly between kernel-HW combinations, the number of operations is a poor and misleading proxy for both latency and energy metrics [28], although it is easy to calculate and therefore often used anyway.

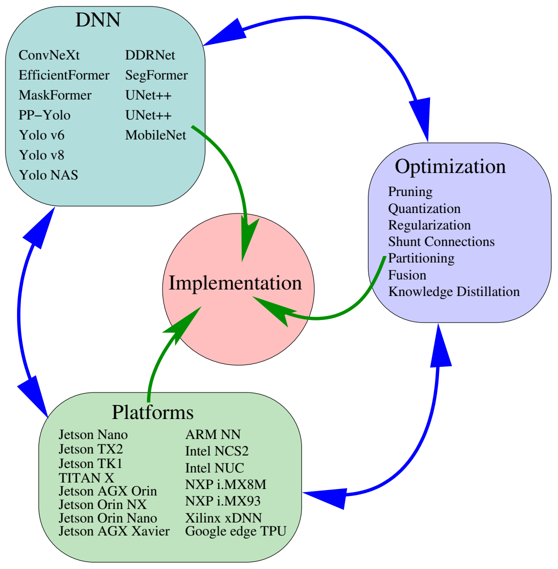


Fig. 6. Design space.

Even for a given algorithm-HW combination, the dynamic settings during execution can have profound effects on energy consumption and latency. In a study of the Nvidia Jetson Nano - MobileNet combination we found that activating four cores increases the energy consumption by $7.5\,\%$ but reduces the latency only by $0.1\,\%$ [29]. Reducing frequency by 4x (from $1479\,\mathrm{MHz}$ to $307\,\mathrm{MHz}$) decreases latency by 2x but energy consumption only by $20\,\%$. Curiously, energy consumption increases again at lower frequencies and thus, there is an optimal frequency of $307\,\mathrm{MHz}$ in a range of $102\,\mathrm{MHz}$ to $1478\,\mathrm{MHz}$ [29].

In every study we conducted [28], [30]–[35] we found unexpected and often surprising relations, giving witness to a complex, highly non-linear design space where small changes in parameters can have over-proportional effects.

Considerable effort has been spent to analyze and understand energy usage and delay behavior of DNNs executing on various platforms, aiming at the granularity of individual layers.

In a recent paper [28] we presented our profiling methodology for assessing the energy efficiency of neural network accelerators at both layer and network granularity. The approach involves extracting per-layer timing reports from recorded power profiles. The power and energy consumption of three prominent neural network accelerators, namely the *Intel Neural Compute Stick 2*, the *Coral Edge TPU*, and the *NXP i.MX8M Plus* is evaluated for three different DNNs using this method. The study investigates the relationship between decreasing sampling frequencies and the average error, as well as the detailed energy consumption of individual DNN layers and layer types.

We found that latency is a better predictor for both overall and dynamic energy than the number of operations per layer, with errors of 10 % and 100 %, respectively. The main conclusions are: a sampling frequency of 200 kHz is necessary to achieve an average error of 5 %; the number of operations is an inadequate predictor of energy consumption; and specific hardware settings significantly influence power and energy consumption, emphasizing the need for their consideration in estimation.

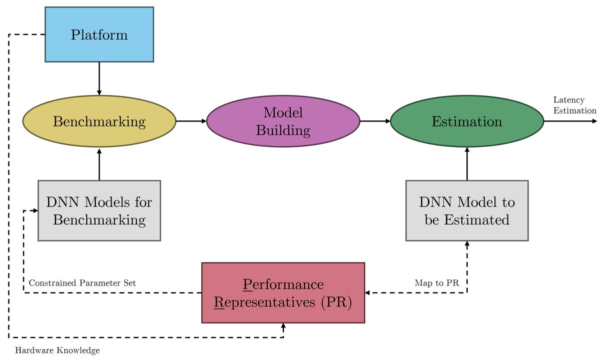

Platform
Benchmarking
Model Building
Estimation
Latency Estimation
DNN Models for Benchmarking
DNN Model to be Estimated
Performance Representatives (PR)
Constrained Parameter Set
Map to PR
Hardware Knowledge

Fig. 7. Overview of the estimation flows of ANNETTE and Blackthorn using selected benchmarking and model building. Optional usage of PRs to further constrain the benchmark parameter space, shown with dashed arrows.

Using the many insights from our profiling experiments ANNETTE [36], [37] and Blackthorn [27] tools have been developed at TU Wien. They perform layer-wise latency estimation of a given DNN for a given platform. ANNETTE uses mixed analytic-stochastic models to estimate the latency of individual or fused layers on CPU-, FPGA-, and NPU-based platforms. Blackthorn uses step-wise linear analytic functions for latency estimations on GPUs and NPUs. ANNETTE and Blackthorn differ (i) by their benchmarking strategy, (ii) the estimation models, and (iii) the targeted platforms. Blackthorn assumes that the estimation functions are a step-wise linear. This assumption has a strong influence on the model building algorithm and the benchmarking strategy, which tries to select measurement points at the edges of the steps in the estimation functions. Natural target platforms for Blackthorn are hardware architectures with many computing resources organized in parallel like GPU based processors and many DNN accelerators. When a platform has been benchmarked and characterized, both estimators have execution times between 0.5 ms and 4 ms, and exhibit estimation errors below 10 %. Table I lists the main features.

For a user of an estimation tool it is important to obtain a good estimate and an assessment of the quality of the estimate, which we call confidence. In the context of ANNETTE we have proposed a confidence metric and a method to quantify the accuracy of the latency estimation. The metric is based on the observation that an estimate is better if it is close to an area with many measurements, as opposed to a point that is hardly covered my actual measurements.

As stated before, the benchmarking process can be very time-consuming, especially depending on the amount of required training data points for the stochastic models of ANNETTE. Therefore, the University of Tübingen has developed a performance modeling methodology and benchmarking strategy, building upon the ANNETTE tool, which aims at reducing the amount of required training data points [38]. Similar to the approach of Blackthorn, the step-wise execution time behavior of many AI accelerator platforms is exploited by restricting the benchmarking parameter space to only one (the last) point of a step. This point we call a Performance Representative (PR). The PRs can either be deduced from the hardware and mapping parameters of an AI accelerator architecture or determined algorithmically from initial parameter sweeps (see Fig. 8).

Sampling the training data only from this restricted parameter set of PRs and mapping target layers to the corresponding PR before making the estimation, leads to a significant reduction in benchmark points, and consequently benchmark time, while maintaining estimation accuracy. How this approach integrates with the existing ANNETTE workflow is shown in Fig. 7 with dashed arrows. Fig. 9 shows the Mean Absolute Percentage Error (MAPE) of a single Conv2D layer estimation for the Versatile Tensor Accelerator (VTA) [39] for different training dataset sizes. Sampling the benchmarks from the set of PRs and mapping the target layer to the corresponding PR and using it to make the estimation outperforms an uninformed sampling approach while needing less than 10 000 training samples.

TABLE I
FEATURES OF THE ESTIMATORS ANNETTE AND BLACKTHORN.

	Benchmarking strategy	Model building	Target platforms
ANNETTE	micro kernels, multi-layer	mixed analytic stochastic	CPUs, FPGAs, NPUs
Blackthorn	micro kernels using dynamically selected measurements	linear and step-wise functions	GPUs, NPUs

TABLE II
ESTIMATION ACCURACY OF ANNETTE AND BLACKTHORN TOOLS.

Network	**Estimation Error [%]**			
	NCS2	**ZCU102**	**Jetson Nano**	**Jetson TX2**
YoloV3	4.1	3.2	-	-
MobileNetV2	4.3	4.2	3.6	4.2
ResNet50	8.2	1.2	2.4	4.8
FPN Net	9.3	7.5	-	-
AlexNet	5.2	4.8	5.5	6.6
VGG16	11.3	6.2	0.5	1.4

V. MULTI-LEVEL PERFORMANCE ESTIMATION OF MULTI-INSTANCE AI COMPUTE PLATFORMS

The very dynamic development in machine learning and AI implies an equally dynamic development of AI hardware platforms, which are composed of multiple NPU and SIMD

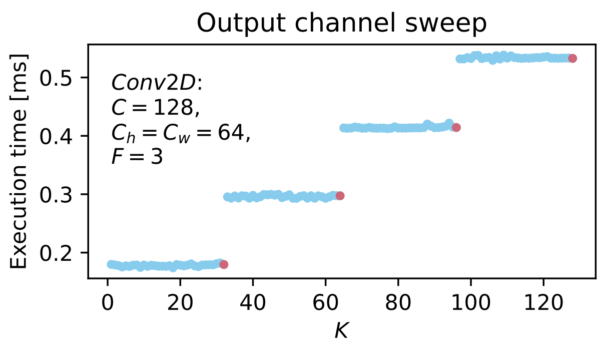


Fig. 8. Step-wise runtime behavior of the Jetson AGX Xavier GPU for increasing Conv2D parameter output channels K. PRs shown in red. [38]

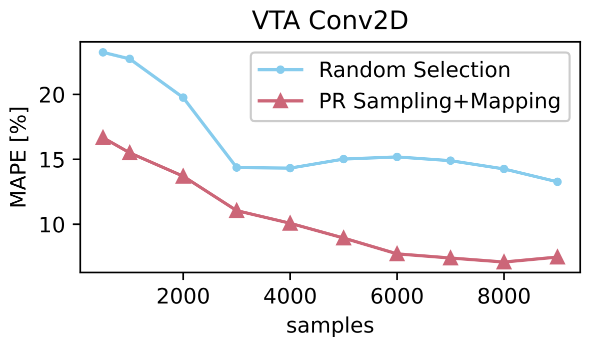
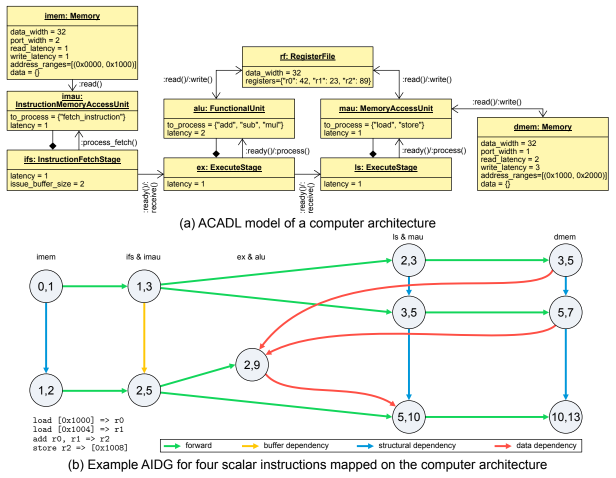


Fig. 9. Mean Absolute Percentage Error (MAPE) of single layer execution time estimation for different training dataset sizes, comparing random sampling from the complete Conv2D parameter space vs. PR sampling and mapping. [38]

cores and a domain-specific memory hierarchy. New AI features are typically designed and trained by AI experts, and then optimized for an embedded target using hardware-aware neural-architecture search (NAS) and mixed deployment, taking into account the heterogeneity of the provided compute instances. To account for the influence of memory accesses between multiple compute instances, the multi-dimensional data flow between across compute instances as well as the shape of the data access pattern that the data flow follows are calculated in order to estimate the amount of data to be transferred through different memory instances in a given time interval. Using the Abstract Computer Architecture Description Language (ACADL) [40] we can build timing models for AI accelerator cores on different abstraction levels. Fine-grained models are based on scalar operations such as addition or multiplication, while coarse grained models employ vector and tensor operations. Given an ACADL model of an AI accelerator core and a DNN mapping, in the form of a loop nest containing a sequence of scalar or tensor instructions, we propagate each instruction through the ACADL model and track the structural, data, and buffer fill level dependencies in the Architectural Instruction Dependency Graph (AIDG) [41]. An example ACADL model together with an AIDG for this model is presented in Fig. 10.



```
load [0x1000] => r0
load [0x1004] => r1
add r0, r1 => r2
store r2 => [0x1008]
```

Fig. 10. Example computer architecture modeled using ACADL and the corresponding AIDG for four instructions mapped onto the ACADL model. [41]

By analyzing the AIDG, we can estimate the end-to-end latency of a DNN mapped onto an AI accelerator core. While it is possible to propagate all instructions of all iterations of a DNN mapping through an ACADL model to construct an AIDG it is not necessary to get an accurate estimation. Because DNN layers are implemented as nested loops with very regular control flow and memory access patterns we only need to analyze a prolog of at maximum 1% of all iterations in an AIDG until the latency $\Delta t_{\text{iteration}}$ and the overlap $\Delta t_{\text{overlap}}$ of consecutive iterations converge to a fix point where the latencies only change in a very small range. Using those two latencies and the latency of prolog Δt_{prolog} containing n_{prolog} iterations we can calculate the end-to-end latency of n iterations in a DNN layer

$$\Delta t = \Delta t_{\text{prolog}} + (n - n_{\text{prolog}}) \cdot (\Delta t_{\text{iteration}} - \Delta t_{\text{overlap}}).$$

This enables us to make fast and very accurate performance estimations for different architectures such as UltraTrail [42] and Gemmini [43] with a MAPE between 0.0001% and 9.78%, compared to a cycle-accurate RTL simulation, that take at maximum 38s to estimate the end-to-end latency of a whole DNN such as AlexNet.

The AIDG allows us to not only accurately estimate the performance of an AI accelerator core but also inject call back functions when a memory is accessed and call a cache or DRAM timing model or a performance model of another AI accelerator core. This provides the possibility to combine various performance estimation methods into a heterogeneous performance model for multi-instance AI compute platforms with shared memory.

Fig. 11(a) shows an example AI compute platform composed out of a RISC-V MCU acting as a controller, two systolic arrays for processing GEMM and convolutional layers, and a SIMD Unit for pooling and activation layers all connected through a shared cache hierarchy and a shared DRAM. This AI compute platform allows for processing DNN layers in

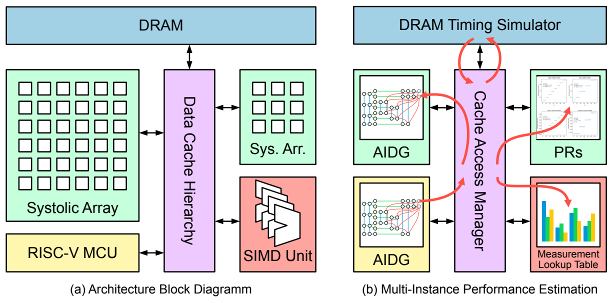


Fig. 11. Combining different performance estimation methods for the performance estimation of multi-instance AI compute accelerator platforms.

parallel or in a pipelined fashion, where one core continuously produces an output feature while another core consumes it.

Using multiple ACADL models with an AIDG analyses together with statistical performance models, measurement bases approaches and a DRAM timing simulator [44] all connected through a cache access manager which simulates cache accesses and latencies (see. Fig. 11(b)) lets us estimate the performance of a whole AI compute platform. First preliminary results using two systolic arrays with a shared cache hierarchy show a MAPE between 0.8% and 12.5% when estimating the performance for a whole DNN mapping.

VI. Summary

We have reviewed optimization and estimation techniques for deploying DNNs on embedded platforms. Specifically, we have discussed state-of-the-art methods for quantizing LLMs and CNNs. For LLMs, the paper describes Activation-aware Weight Quantization (AWQ) and the TinyChat inference system, which significantly reduce model size and enhance inference speed across various platforms. For CNNs we presented pruning techniques and integrated hardware-aware optimization flows, particularly for FPGA and CPU-based platforms. Furthermore, we have reviewed several estimation techniques for predicting the latency of DNNs on specific hardware. Specifically we discussed three estimation approaches: ANNETTE is based on a mixed analytic-stochastic model; Blackthorn is an analytic model based on step-wise linear functions; and an estimation approach based on detailed model of the hardware architecture.

Main conclusions from the presented work are:

- Not all weights in a DNN are equally important and by selective quantization the memory footprint and performance can be greatly improved without sacrificing too much quality. The described Activation-aware Weight Quantization approach allows for significant reduction in model size (up to 4×) and accelerates inference by 3-4×.
- Hardware-aware pruning techniques for CNNs can drastically reduce the computational load and improve inference times on embedded systems. For example, the integration of channel level, layer level, and hardware-aware pruning can lead to significant performance gains with FPGA and RISC-V implementations.
- There are many non-linear relationships between hardware architectures and hardware settings on one hand and DNN algorithms on the other hand. Some of them we have reviewed, like the different implementation efficiencies of different kernels and the case of the Nvidia Jetson Nano, where increasing the number of active cores has a minimal effect on latency but significantly increases energy consumption. These findings underscore the complexity of optimizing DNN performance on embedded platforms, where small changes in hardware settings and algorithms can lead to disproportionate effects on energy and latency.
- Latency estimation tools can greatly speed-up the design space exploration by providing relatively accurate estimates in very short time. The required latency accuracy is about 10% or below, which has been shown to be achievable. There is significant effort to be invested in model building and banchmarking as preparation for the actual estimation. But this effort is quickly amortized if the same broad classes of DNNs and hardware architectures are used in a few application projects.

Overall, the work presented here shows that even high end DNN algorithms can be efficiently deployed on constrained embedded devices, given the appropriate methodology and tools.

References

[1] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, “Llama: Open and efficient foundation language models,” *arXiv preprint arXiv:2302.13971*, 2023.

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.

[3] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, “Gptq: Accurate post-training quantization for generative pre-trained transformers,” *arXiv preprint arXiv:2210.17323*, 2022.

[4] J. Lin, H. Yin, W. Ping, Y. Lu, P. Molchanov, A. Tao, H. Mao, J. Kautz, M. Shoeybi, and S. Han, “Vila: On pre-training for visual language models,” in *CVPR*, 2024.

[5] G. Xiao, J. Lin, M. Seznec, J. Demouth, and S. Han, “Smoothquant: Accurate and efficient post-training quantization for large language models,” *arXiv preprint arXiv:2211.10438*, 2022.

[6] G. Penedo, Q. Malartic, D. Hesslow, R. Cojocaru, A. Cappelli, H. Alobeidli, B. Pannier, E. Almazrouei, and J. Launay, “The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only,” *arXiv preprint arXiv:2306.01116*, 2023.

[7] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

[8] Y. J. Kim, R. Henry, R. Fahim, and H. H. Awadalla, “Who says elephants can’t run: Bringing large scale moe models into cloud scale production,” *arXiv preprint arXiv:2211.10017*, 2022.

[9] H. Li, Z. Wang, X. Yue, W. Wang, H. Tomiyama, and L. Meng, “An architecture-level analysis on deep learning models for low-impact computations,” *Artificial Intelligence Review*, vol. 56, pp. 1971–2010, 2023.

[10] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges,” *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 126–136, 2018.

[11] Y. He and S. Han, “ADC: automated deep compression and acceleration with reinforcement learning,” in *European Conference on Computer Vision (ECCV)*, Munich, Germany, September 2018.

[12] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of Neural Networks using DropConnect," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, Atlanta, GA, USA, June 2013, pp. 1058–1066.

[13] T. Chen, B. Ji, T. Ding, B. Fang, G. Wang, Z. Zhu, L. Liang, Y. Shi, S. Yi, and X. Tu, "Only Train Once: A One-Shot Neural Network Training And Pruning Framework," in *Advances in Neural Information Processing Systems NeurIPS*, online, December 2021, pp. 19 637–19 651.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, June 2016, pp. 770–778.

[15] H. Li and L. Meng, "Hardware-aware approach to deep neural network optimization," *Neurocomputing*, vol. 559, p. 126808, 2023.

[16] T. Chen, B. Ji, D. Tianyu, B. Fang, G. Wang, Z. Zhu, L. Liang, Y. Shi, S. Yi, and X. Tu, "Only train once: A one-shot neural network training and pruning framework," in *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[17] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2095809919306356

[18] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: Accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.

[19] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *CoRR*, vol. abs/2106.08295, 2021. [Online]. Available: https://arxiv.org/abs/2106.08295

[20] Q. Li, H. Li, and L. Meng, "A generic deep learning architecture optimization method for edge device based on start-up latency reduction," *Journal of Real-Time Image Processing*, vol. 21, 2024.

[21] H. Li, Z. Wang, X. Yue, W. Wang, H. Tomiyama, and L. Meng, "A comprehensive analysis of low-impact computations in deep learning workloads," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, New York, USA, June 2021, pp. 385–390.

[22] E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, "GAP-8: A RISC-V SoC for AI at the Edge of the IoT," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4.

[23] Z. Su, Q. Li, H. Kaneko, H. Li, and L. Meng, "Optimization and deployment of dnns for risc-v-based edge ai," in *The 2024 IEEE International Conference on Real-time Computing and Robotics*, Ålesund, Norway, June 2024.

[24] X. Yue, H. Li, and L. Meng, "An ultralightweight object detection network for empty-dish recycling robots," *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1–12, 2023.

[25] X. Yue and L. Meng, "Yolo-msa: A multiscale stereoscopic attention network for empty-dish recycling robots," *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1–14, 2023.

[26] Y. Ge, Z. Li, X. Yue, H. Li, Q. Li, and L. Meng, "Iot-based automatic deep learning model generation and the application on empty-dish recycling robots," *Internet of Things*, vol. 25, p. 101047, 2024.

[27] M. Lechner and A. Jantsch, "Blackthorn: Latency estimation framework for CNNs on embedded Nvidia platforms," *IEEE Access*, 2021. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2021/MartinLechner-IEEEAccess.pdf

[28] M. Wess, D. Dallinger, D. Schnöll, M. Bittner, M. Götzinger, and A. Jantsch, "Energy profiling of DNN accelerators," in *Proceedings of the 26th Euromicro Conference on Digital System Design (DSD)*, Durres, Albania, September 2023. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2023/MatthiasWess-DSD.pdf

[29] A. Glinserer, M. Lechner, and A. Wendt, "Automated pruning of neural networks for mobile applications," in *IEEE International Conference on Industrial Informatics (INDIN)*, 2021.

[30] I. Shallari, I. Sánchez Leal, S. Krug, A. Jantsch, and M. O'Nils, "Design space exploration on IoT node: Trade-offs in processing and communication," *IEEE Access*, 2021. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2021/IridaShallari-IEEEAccess.pdf

[31] I. S. Leal, E. Saqib, I. Shallari, A. Jantsch, S. Krug, and M. O'Nils, "Waist tightening of CNNs: A case study on tiny yolov3 for distributed iot implementations," in *Proceedings of the Real-time And intelliGent Edge computing workshop (RAGE)*, San Antonio, Texas, May 2023. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2023/IsaacSanchezLeal-RAGE.pdf

[32] E. Saqib, I. S. Leal, I. Shallari, A. Jantsch, S. Krug, and M. O'Nils, "Optimizing the IoT performance: A case study on pruning a distributed CNN," in *Proceedings of the IEEE Sensors Applications Symposium (SAS)*, 2023. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2023/EirajSaqib-SAS.pdf

[33] I. S. Leal, I. Shallari, S. Krug, A. Jantsch, and M. O'Nils, "Impact of input data on intelligence partitioning decisions for IoT smart camera nodes," *Electronics*, vol. 10, no. 16, 2021. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2021/IsaacLeal-MDPIElectronics.pdf

[34] S. Holly, A. Wendt, and M. Lechner, "Profiling energy consumption of deep neural networks on Nvidia Jetson Nano," in *2020 11th International Green and Sustainable Computing Workshops (IGSC)*, 2020, pp. 1–6.

[35] A. Wendt, H. Possegger, M. Bittner, D. Schnöll, M. Wess, D. Malić, H. Bischof, and A. Jantsch, "A pedestrian detection case study for a traffic light controller," in *Embedded Machine Learning for Cyber-Physical, IoT, and Edge Computing - Software Optimizations and Hardware/Software Codesign*, S. Pasricha and M. Shafique, Eds. Springer, 2023, pp. 75–96. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-39932-9_4

[36] M. Wess, M. Ivanov, C. Unger, A. Nookala, A. Wendt, and A. Jantsch, "ANNETTE: Accurate neural network execution time estimation with stacked models," *IEEE Access*, vol. 9, pp. 3545–3556, 2021. [Online]. Available: http://jantsch.se/AxelJantsch/papers/2021/MatthiasWess-IEEEAccess.pdf

[37] M. Wess and A. Jantsch, "Confidence for latency estimation of DNN accelerators: A blackbox approach," *Accepted for publication*, 2024.

[38] A. L.-F. Jung, J. Steinmetz, J. Gietz, K. Lübeck, and O. Bringmann, "It's all about PR – Smart Benchmarking AI Accelerators using Performance Representatives," in *2024 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2024.

[39] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A Hardware-Software Blueprint for Flexible Deep Learning Specialization," 2019.

[40] M. M. Müller, A. R. M. Borst, K. Lübeck, A. L.-F. Jung, and O. Bringmann, "Using the abstract computer architecture description language to model ai hardware accelerators," in *MBMV 2024; 27. Workshop*, 2024, pp. 19–30.

[41] K. Lübeck, A. L.-F. Jung, F. Wedlich, M. M. Müller, F. N. Peccia, F. Thömmes, J. Steinmetz, V. Biermaier, F. Adrian, P. P. Bernardo, and O. Bringmann, "Automatic generation of fast and accurate performance models for deep neural network accelerators," *Accepted for publication in ACM Transactions on Embedded Computing Systems (TECS)*, 2024.

[42] P. P. Bernardo, C. Gerum, A. Frischknecht, K. Lübeck, and O. Bringmann, "UltraTrail: A configurable Ultralow-Power TC-ResNet AI Accelerator for Efficient Keyword Spotting," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4240–4251, 2020.

[43] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[44] L. Steiner, M. Jung, F. S. Prado, K. Bykov, and N. Wehn, "Dramsys4.0: An open-source simulation framework for in-depth dram analyses," *International Journal of Parallel Programming*, vol. 50, no. 2, p. 217–242, Mar. 2022. [Online]. Available: http://dx.doi.org/10.1007/s10766-022-00727-4