Neural Architecture Search for low-power MCUs

Alessio Burrello alessio.burrello@unibo.it
alessio.burrello@polito.it
Daniele Jahier Pagliari
Matteo Risso
Beatrice Alessandra Motetti

PULP Platform
Open Source Hardware, the way it should be!
DNNs at the Extreme Edge

- Near-sensor DNN inference has several potential benefits w.r.t. a traditional cloud-centric approach:
  1. More predictable and lower (*) latency
  2. Data privacy
  3. Lower energy consumption (*)

(*) possibly
The drone follows the head of the human.
Deep Neural Network Architecture Search for Accurate Visual Pose Estimation aboard Nano-UAVs

E. Cereda, L. Crupi, M. Risso, A. Burrello, L. Benini, A. Giusti, D. Jahier Pagliari, and D. Palossi

Reality → Expectation

What changed from reality to expectation? Neural Architecture Search

Frontnet [2]
#Params: 304k
#MACs/inference 14.7M
Max throughput 45.3 FPS
MAE → x-axis 0.33
 y-axis 0.12
 angle 0.77
Mission → Failed

NAS network [1]
#Params: 65k
#MACs/inference 7.4M
Max throughput 51.2 FPS
MAE → x-axis 0.25
 y-axis 0.11
 angle 0.52
Mission → Complete


DNNs Deployment Flow

Training-time
- Neural Architecture Search (NAS)
- Pruning
- Mixed-Precision Search
- Quantization-Aware Training (QAT)

Post-training
- Post-Training Quantization
- AI Compilation
  - Graph Rewriting
  - Memory Tiling
  - Primitives Selection

Run-time
- Collaborative Inference
  - Adaptive Inference
    - Big/little
    - Slimmable
    - Multi-precision
    - Hierarchical
- Optimized Binary for Target

DNNs Deployment Flow Diagram

- TensorFlow
- PyTorch
- ONNX
- DNN “Seed”
- Dataset
- HW Model
- Trained Model (HDF5, Tflite, ONNX,..)
- Compiled Model (C, C++, FlatBuffer)

DNNs Deployment Flow

- DNNs Deployment Flow
  - Post-training AI Compilation
  - Run-time Collaborative Inference
    - Adaptive Inference
      - Big/little
      - Slimmable
      - Multi-precision
      - Hierarchical
  - Optimized Binary for Target

DNNs Deployment Flow

- DNNs Deployment Flow
  - Post-training AI Compilation
  - Run-time Collaborative Inference
    - Adaptive Inference
      - Big/little
      - Slimmable
      - Multi-precision
      - Hierarchical
  - Optimized Binary for Target

DNNs Deployment Flow

- DNNs Deployment Flow
  - Post-training AI Compilation
  - Run-time Collaborative Inference
    - Adaptive Inference
      - Big/little
      - Slimmable
      - Multi-precision
      - Hierarchical
  - Optimized Binary for Target

DNNs Deployment Flow

- DNNs Deployment Flow
  - Post-training AI Compilation
  - Run-time Collaborative Inference
    - Adaptive Inference
      - Big/little
      - Slimmable
      - Multi-precision
      - Hierarchical
  - Optimized Binary for Target

DNNs Deployment Flow

- DNNs Deployment Flow
  - Post-training AI Compilation
  - Run-time Collaborative Inference
    - Adaptive Inference
      - Big/little
      - Slimmable
      - Multi-precision
      - Hierarchical
  - Optimized Binary for Target

DNNs Deployment Flow

- DNNs Deployment Flow
  - Post-training AI Compilation
  - Run-time Collaborative Inference
    - Adaptive Inference
      - Big/little
      - Slimmable
      - Multi-precision
      - Hierarchical
  - Optimized Binary for Target
DNNs Deployment Flow

Training-time
- Neural Architecture Search (NAS)
- Pruning
- Mixed-Precision Search
- Quantization-Aware Training (QAT)

Post-training
- AI Compilation
  - Graph Rewriting
  - Memory Tiling
  - Primitives Selection

Run-time
- Collaborative Inference
  - Big/little
  - Slimmable
  - Multi-precision
  - Hierarchical

Optimized Binary for Target
2. (Differentiable) Neural Architecture Search
Neural Architecture Search

• **Motivation:** Picking hyper-parameters manually is tricky
  • Biases (rules of thumb, traditions, etc.)
  • Fragmented and coarse design space explorations (e.g., width/res mult in MobileNets)
  • Classic ML: hand-craft features, DL: hand-craft feature extractors!

• **Neural Architecture Search (NAS)**
  • Automatic optimization of the network topology, exploring a large and fine-grain design space of hyper-parameter settings
  • Typically **multi-objective**: co-optimize accuracy and model complexity
    • Model size/#MACs....
    • ...or better, **latency/energy directly** (requires models)!

ETH Zürich
Classic NAS

• Key steps:
  1. Define the search space
  2. Define a search engine
  3. Build a performance estimator

• Thousands of GPU-hours per search!

• Procedure:

  Guess
  Train
  Evaluate

Propose 1+ new architecture(s)

Feedback to drive the search
Differentiable NAS (DNAS)

• Relax the search space to make it continuous and differentiable

• Optimize the topology by gradient descent during training

• Reduce search costs: Gradient-based optimization is much more lightweight than black-box methods (RL or Evolutionary)
3. PLiNIO: Plug-and-play Lightweight Neural Inference Optimizer
PLiNIO Motivation

**SUPERNET:** coarse-grain layer type selection

**PIT:** fine-grain layer’s hyper-parameters selection

**MIXPREC:** precision assignement

Developed by us
PLiNIO is a Python package built on-top of the PyTorch ecosystem that provides a Plug-and-play Lightweight tool for the Inference Optimization of DNNs.

PLiNIO exploits as main optimization engine DNAS algorithms which notoriously balance flexibility and lightness.
PLiNIO is open-sourced on github
PLiNIO (cont’d)

• PLiNIO allows to automatically optimize your DNN's architecture with *no more than three additional lines of code* to your original training loop.

```python
# A plinio-enhanced pytorch training loop
model = ResNet()
model = plinio.PIT(model, input_shape=(C, H, W))  # 1 Convert the model
for epoch in range(N_EPOCHS):
    for sample, target in data:
        output = model(sample)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

1. Convert the model
2. Compute additional loss
3. Export the optimized model
3. Developed Differentiable NAS algorithms
PIT: Pruning in Time

- Search space: For each Convolutional or Fully-Connected layer

- Smaller number of output channels
- Smaller receptive field
- Larger dilation factor
PIT: Pruning in Time

- Add a **L1 regularization term** to the training loss function that brings masks to 0
  - More 0-valued masks $\rightarrow$ smaller network
- Classical regularizers:
  - **N. of weights**, correlates with memory occupation
  - **N. of MACs**, correlates with latency/energy
- HW regularizers: piece-wise polynomial functions

Future: **GAP9, Occamy** and many others...

**Final Loss Function:**

$$
\min_{W, \theta} \mathcal{L}(W; \theta) + \lambda R(\theta)
$$

Regularizer, function of Trainable binary masks
PIT: Results

- 4 edge-relevant benchmarks (biosignals, keyword spotting).
- Up to 8x smaller and 7x faster models at iso-performance
PIT into the wild

- PIT has been now extended to 2D networks for vision.
  - Example: drone-to-human pose estimation in low-power nanodrones
- **Same results** of previous hand-tuned network with *3x less memory*, thanks to PIT
- Collaboration with POLITO + UNIBO + ETHZ + IDSIA (Lugano) → presented @ ICRA23
Multi-Regularization Loss

• From the designer perspective the main goal is finding an optimal trade-off between accuracy and complexity while satisfying the memory requirements $s^*$ of the target.
• We develop a novel multi-regularization loss formulation:

\[
\mathcal{L}_{\text{task}} + \mathcal{L}_{\text{reg}} = \lambda |S(\theta) - s^*| + \mu O(\theta)
\]

Size-Loss

OPs-Loss

Zero when the target is met

Used as proxy of energy consumption

• The mutual importance of regularization loss terms is controlled with $\lambda$ and $\mu$:
  • $\lambda$ is fixed and such as to satisfy $\lambda >> \mu$
  • $\mu$ is tweaked to explore different Accuracy vs Energy tradeoffs
Multi-Regularization: Results

- Experiments on three edge relevant use-cases from MLPerf Tiny Benchmark Suite which proposes reference optimized network implementations.

- We obtain rich Pareto sets of architectures in the **OPs vs. accuracy space**, with memory footprints spanning from **75%** to **6.25%** of baseline networks.
Other works

• Multi-constraint loss: a new NAS formulation to respect both a memory constraint and a maximum latency

• Multi-precision search: a different precision can be chosen for each channel of a tensor using gradient-descent.
  • We support the export of quantized networks which can be imported from DORY (Deeploy soon) and executed on PULP successfully!!!

• Heterogeneous-NAS: NAS for heterogenous hardware. It maps different part of a layer to different accelerators
  • Optimize network during NAS based on the type of layer/precision supported by each accelerator in a heterogeneous SoC
  • Tested on DIANA AIMC and Digital Accelerators
  • Accepted at ISLPED2023
What’s next?

- Adding new hardware models to improve the NAS search (GAP9, Occamy...)

- Insert the hardware in the loop to have a precise feedback of the network on the MCU

- Targeting full application, trying to optimize a task and not only a loss

- Extend PlinIO to include all methods and allow for automatic end-to-end optimization pipelines

- Interface the NAS tools with the deployment pipelines
Neural Network Deployment on Heterogeneous Systems

Moritz Scherer
scheremo@iis.ee.ethz.ch
Neural Network Deployment

• Until very recently, residual CNNs dominated the state-of-the-art
  • ResNets
  • MobileNets (v1, v2, v3, ...)
  • EfficientNets
• Dory was specifically designed for integer-quantized residual CNNs
  • Support for two concurrent branches
  • Support for integer arithmetic on the PULP Cluster
  • Support for memory-aware layer-wise tiling
  • Efficient parallelization strategies for various operators

• A match that led to advancements in the SoA several times over!
Dory for Deployment – Challenges & Limitations

• Dory deployment with accelerators is challenging
  • Some layers have very low arithmetic intensity
    • Depthwise convolutions, Matrix multiplications, ...
  • Depth-first tiling helps to keep execution compute bound
    • Siracusa: Executing IRB layers depth-first improves MobileNetv2 performance by 60%!

• Even more challenges for our deployment tools
  • Transformers dominate all ML benchmarks
  • Low-precision floating point training & inference on microcontrollers is gaining traction
  • Occamy & MemPool are breaking ground on HPC PULP systems
Deeploy – Enabling Heterogeneous Deployment

Context-Free Templates

Expressive Data Types

Flow- and Pass-based Graph Editing

Flexible Operator Offloading

Self-Containing Engines
Deeploy – Context-Free Templates

• What does it take to run a convolution?
  • Inputs & weights need to be pre-allocated
  • Kernel templates need to run on all cores
  • Outputs need to be moved back

• Deeploy uses context-free templates
  • DMA calls, etc. are generated by Deeploy
  • Only kernel calls need to be implemented
Deeploy – Expressive Data Types

• Deeploy uses expressive primitive types
  • Immediate, Pointer, Struct & Future

• Bring your own immediate types
  • Only need to implement a function that checks a value
  • Compose your own types in pointers, structs, and futures

• Automatic strong type checking
  • For your own immediate types, and all composed types
Deeploy – Self-Contained Engines

• The PULP SoC is designed for adding accelerators
  • General-Purpose Accelerators like the PULP Cluster
  • Application-specific Accelerators like N-EUREKA, NE16, CUTIE, ITA, ...

• Compute engines are highly customizable
  • Data types, Programming model, Memory access

• Deeploy keeps each engine self-contained
  • Engine-specific, context-free templates, programming model, and data types
  • The same engine in a different SoC works the same
Deeploy – Simple Microcontrollers

- This lets us generate network inference code!

```
DeeployNetwork_Deeploy_BUFFER_output_0_ctxt = (cmsis_nn_context){.buf = NULL, .size = 0};
DeeployNetwork_Deeploy_BUFFER_output_0_activation = (cmsis_nn_activation){.min = -64, .max = 63};
DeeployNetwork_Deeploy_BUFFER_output_0_fc_params = (cmsis_nn_fc_params){
    .input_offset = 0, .output_offset = -64, .filter_offset = 0, .activation = {.min = -64, .max = 63}};
DeeployNetwork_Deeploy_BUFFER_output_0_quant_params = (cmsis_nn_per_tensor_quant_params){.multiplier = 9609216, .shift = 0};
DeeployNetwork_Deeploy_BUFFER_output_0_input_dims = (cmsis_nn_dims){.n = 1, .h = 1, .w = 1, .c = 512};
DeeployNetwork_Deeploy_BUFFER_output_0_filter_dims = (cmsis_nn_dims){.n = 512, .h = 1, .w = 1, .c = 10};
DeeployNetwork_Deeploy_BUFFER_output_0_output_dims = (cmsis_nn_dims){.n = 1, .h = 1, .w = 1, .c = 10};
DeeployNetwork_DeeployBUFFER_output_0_bias_dims = (cmsis_nn_dims){.n = 1, .h = 1, .w = 1, .c = 10};

arm_fully_connected_s8(
    &DeeployNetwork_Deeploy_BUFFER_output_0_ctxt, &DeeployNetwork_Deeploy_BUFFER_output_0_fc_params,
    &DeeployNetwork_Deeploy_BUFFER_output_0_quant_params, &DeeployNetwork_Deeploy_BUFFER_output_0_input_dims,
    DeeployNetwork_Deeploy_BUFFER_28, &DeeployNetwork_Deeploy_BUFFER_output_0_filter_dims,
    DeeployNetwork_Deeploy BUFFER_32, &DeeployNetwork_Deeploy BUFFER output_0 bias dims,
    DeeployNetwork_Deeploy BUFFER_classifier__QL_REPLACED_INTEGERIZE_UNSIGNED_ACT_PASS_0_add,
    &DeeployNetwork_Deeploy BUFFER output 0 output dims, DeeployNetwork_Deepl
```
Deeploy – Flexible Operator Offloading

• But only for single-core, single-memory-level systems
  • Everything happens in the same execution context

• To run on a PULP Cluster, we have to
  • Move memory with the DMA
  • Offload code to the cluster

• From the Fabric Controller’s POV
  • The DMA and Cluster work asynchronously

We need a way to model offloading and concurrent execution
Deeploy – Closures and Futures

- Deeploy uses closures to offload kernels
  - A closure is a function that wraps a *kernel call* and its *state*
- Asynchronous computation produces *Future*-typed outputs
  - Futures are values that “will be available later”
  - Before generating a Future, we need to *dispatch* it
  - Before accessing a Future, we need to *resolve* it
  - Future types provide code to dispatch and resolve
- Futures enable local synchronization
  - No OS, tasks or threads required – but supported

Futures allow us to address engines concurrently
Deeploy – Tiling & Graph Manipulation

• Deeploy comes with a flexible & powerful graph editing framework

• Passes are used for match-based transformations
  • “Replace all occurrences of A->B with C”

• Flows are used for graph-level information propagation
  • Tensor type inference
  • Bias pushing
  • Tensor liveness analysis

• Deeploy’s tiling algorithm combines passes and flows
  • And allows for depth-first tiling, as well!
Deeploy – Tile Constraint Flow

- We find our pattern with a pass
- Tiling constraints are computed with a flow
  - Constraints(B) = PW-Constraints(Constraints(A))
  - Constraints(C) = DW-Constraints(Constraints(B))
  - Constraints(D) = PW-Constraints(Constraints(C))
  - Constraints(E) = Addition-Constraints(Constraints(A), Constraints(D))
- Using ORTools, we can compute a correct tiling strategy
Deeploy – Graph Tiling

• With our tiling solution, implement a replacement pass
  • Duplicate subgraph
  • Add memory transfer nodes
  • And the rest of the framework manages code generation!

1. Duplicate subgraph
2. Add memory transfer nodes
3. And the rest of the framework manages code generation!
Deeploy – Ongoing and Future Work

• Engine support is growing, and an open-source release is on the horizon
  • Implemented: ARM Cortex-M, MemPool, ITA, PULP Cluster
  • WIP: N-EUREKA, Floating point support, ...
  • Future Work: Multi-Cluster systems like Occamy, Carfield, ...

• Deeploy is designed with extensions in mind
  • Flows, Futures, Closures, and Tiling were designed as extensions
  • New engines and systems are crucial and easy to get started on

• We are looking for contributors!
  • Talk to me, Victor, Francesco, or Alessio – there’s plenty to do!
Thanks for the attention