Challenges for Embedded Machine Learning on Custom RISC-V ASIPs

Philipp van Kempen

Technical University of Munich
TUM School of Computation, Information and Technology
Chair of Electronic Design Automation

Munich, 23th April 2024

This work has been developed in the project MANNHEIM-FlexKI. MANNHEIM-FlexKI is funded by the German Ministry of Education and Research (BMBF) (reference numbers: 01IS22086A-L). The authors are responsible for the content of this publication.
Content

1. Motivation / Scenario
2. Overview of Challenges
3. Solutions
4. Conclusion / Outlook
Embedded Machine Learning

- Machine learning workloads are emerging rapidly → Resource demands are rising exponentially

- Conventional (cloud-based) approach has lots of disadvantages
  - Power consumption (CO2 footprint)
  - Latency (not suitable for real-time applications)
  - Privacy

→ Run Inferences on-device

- Focus: Extreme Edge Devices (TinyML)
  - Many methodologies could also be applied to more powerful HW later
ASIP Design

• There exist numerous off-the-shelf platforms for embedded ML
  • Edge TPU (Google)
  • Jetson Nano (NVIDIA)

• For low-power applications (MCUs) these are not applicable
  ▪ General purpose MCUs are lacking support for efficient TinyML operations
  ▪ Specialize hardware (ISA+Microarchitecture) to targeted application
    → Design Application Specific Instruction-Set Processors (ASIPs)

• Approaches:
  ▪ Manual
  ▪ HW/SW-Codesign driven (High-level Synthesis)
  ▪ Mixed
Our Vision

- **ISA**
  - RISC-V Extensions
  - Custom Instructions

- **Model**
  - Network Topology
  - Training
  - Compression

- **HW**
  - Microarchitecture
  - Memory Hierarchy
  - RTL Design

- **SW**
  - ML Deployment
  - Kernel Libraries
  - Compiler Optimizations

- **Co-Design**

ISA DSE

HW DSE

ISA-aware NAS

HW-aware NAS

Autotuning
# Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td></td>
</tr>
<tr>
<td>Benchmarking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profiling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retargeting</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Philipp van Kempen (TUM) | 23.04.2024
Challenge: Availability

**Problem:** Hardware for testing on-device is not available (or yet to be designed)
- Example: RISC-V Vector Extension (RVV v1.0)
  - Ratified in Late 2021, ramp-up for HW using RVV takes at least 2-3 years
  - Numerous HW released with old specification → Avoid if possible
  - First commercial development board showed up in early 2024

**Solution:** Virtual Prototyping
- [ETISS]: Extendable Translating Instruction Set Simulator
- [CoreDSL]: Describe processor cores at the level of their instruction set architecture
- [CorePerfDSL]: Modelling of pipeline/microarchitecture for performance estimation
# Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td>Virtual Prototyping (ETISS &amp; CoreDSL)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparison between different Frameworks, Targets,… difficult</td>
<td></td>
</tr>
<tr>
<td>Profiling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retargeting</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenge: Benchmarking

**Problem:** Running comparable benchmarks on X targets, Y frameworks, Z toolchains, U Models
- Need a single tool for each target/framework/toolchain/model instead of hardcoded scripts

**Solution:** End-to-End TinyML Deployment and Benchmarking Flow
- **[MLIF]** (Machine Learning Interface)
  - Framework/target-independent abstraction layers for Target SW
- **[MLonMCU]**
  - Provides support for
    - 15+ targets (mainly RISC-V simulators)
    - 6 backends ([TVM] and TFLM)
  - Handling of Dependencies
  - Analysis and Exploration methods
  - Designed with parallelism/reproducibility in mind
# Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td>Virtual Prototyping (ETISS &amp; CoreDSL)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparison between different Frameworks, Targets,… difficult</td>
<td>TinyML Deployment &amp; Benchmarking Flow (MLonMCU)</td>
</tr>
<tr>
<td><strong>Profiling</strong></td>
<td>Where to apply optimizations? How to find bottlenecks?</td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retargeting</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenge: Profiling

**Problem:** Do not waste time optimizing irrelevant parts of a program

- **Example:**
  - Optimized NN-kernel to be 2x faster → End-to-End inference speed only improves by 1%?
  - 95% of the total runtime is spent in different layers

**Solution:** Multi-level profiling/tracing methodology and bottleneck detection

- Convert RTL/ISS traces into Gprof/Callgrind compatible format → Allows to use existing tools for analysis
- Multiple Events: Instructions, Cycles, Cache Misses, Branches,…
- Annotation of sources at various abstraction layers → ML Layers, C/C++, LLVM-IR, Assembly

Work in progress!
### Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td>Virtual Prototyping (ETISS &amp; CoreDSL)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparison between different Frameworks, Targets,… difficult</td>
<td>TinyML Deployment &amp; Benchmarking Flow (MLonMCU)</td>
</tr>
<tr>
<td>Profiling</td>
<td>Where to apply optimizations? How to find bottlenecks?</td>
<td>Multi-level profiling based on static/dynamic binary/trace analysis</td>
</tr>
<tr>
<td>Validation</td>
<td>How to guarantee Model accuracy is maintained during deployment?</td>
<td></td>
</tr>
<tr>
<td>Tuning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retargeting</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Challenge: Validation**

**Problem:** We have to maintain original model’s accuracy during deployment

- Degradations in accuracy can happen at several stages
  - Compression (Quantization/Pruning)
  - ML Compiler / Kernels → Rounding, Overflows,…
  - Bugs (SW Compiler, ISA, RTL)

**Solution:** Dataset-based automatic validation framework integrated in MLonMCU

- Target-independent (MLIFIO abstraction layer)
- Highly configurable (Supported Metrics, Thresholds,…)
- Automated and efficient (minimal runtime/resource overheads)
# Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td>Virtual Prototyping (ETISS &amp; CoreDSL)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparison between different Frameworks, Targets,… difficult</td>
<td>TinyML Deployment &amp; Benchmarking Flow (MLonMCU)</td>
</tr>
<tr>
<td>Profiling</td>
<td>Where to apply optimizations? How to find bottlenecks?</td>
<td>Multi-level profiling based on static/dynamic binary/trace analysis</td>
</tr>
<tr>
<td>Validation</td>
<td>How to guarantee Model accuracy is maintained during deployment?</td>
<td>Target-independent Validation Flow in MLonMCU (MLIFIO)</td>
</tr>
<tr>
<td>Tuning</td>
<td>Can we optimize the generated Kernels for specific targets efficiently?</td>
<td></td>
</tr>
</tbody>
</table>

Retargeting
Challenge: Tuning

Problem: How to get target-optimized ML kernels
  • Default (fallback) TVM kernels should be avoided
  • Writing hand-optimized kernels (see CMSIS-NN) is infeasible

Solution: Perform on-device autotuning with TVM
  • Available tuners: AutoTVM, AutoScheduler, MetaSchedule
  • Improvements required to improve the tuning process
    • Reliability
    • Cost Models (target-aware & workload-aware)
    • Efficiency
# Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td>Virtual Prototyping (ETISS &amp; CoreDSL)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparison between different Frameworks, Targets,… difficult</td>
<td>TinyML Deployment &amp; Benchmarking Flow (MLonMCU)</td>
</tr>
<tr>
<td>Profiling</td>
<td>Where to apply optimizations? How to find bottlenecks?</td>
<td>Multi-level profiling based on static/dynamic binary/trace analysis</td>
</tr>
<tr>
<td>Validation</td>
<td>How to guarantee Model accuracy is maintained during deployment?</td>
<td>Target-independent Validation Flow in MLonMCU (MLIFIO)</td>
</tr>
<tr>
<td>Tuning</td>
<td>Can we optimize the generated Kernels for specific targets efficiently?</td>
<td>TVM’s MetaScheduler with improvements</td>
</tr>
<tr>
<td>Retargeting</td>
<td>How to avoid manual efforts to support new target hardware?</td>
<td></td>
</tr>
</tbody>
</table>
Retargeting

Definition

“In software engineering, retargeting is an attribute of software development tools that have been specifically designed to generate code for more than one computing platform.”

→ Here: Retargetable Compilers

Types of Compilers

• SW Compilers (LLVM, GCC) → See next slides
• ML Compilers (TVM) → Planned
• HW Compilers
Introducing Seal5

Seal5 - Semi-automated LLVM Support for RISC-V ISA Extensions (Including Autovectorization)

Inputs
- CoreDSL code for custom instructions
- Optional: YAML Settings

Outputs
- Patched LLVM Toolchain

```c
CV_SDIOTSP_H {
    encoding: 5'b10101 :: 1'b0 :: 1'b0 :: rs2[4:0] :: rs1[4:0] :: 3'b000 :: rd[4:0] :: 7'b1111011;
    assembly: "{name(rd)}, {name(rs1)}, {name(rs2)}";
    behavior: {
        if (rd != 0) X[rd] = (unsigned<32>)(signed)(
            X[rd] +
            ((signed)X[rs1][15: 0] * (signed)X[rs2][15: 0]) +
            ((signed)X[rs1][31:16] * (signed)X[rs2][31:16]));
    }
}
```
## Retargeting Support Levels (LLVM)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Assembler</th>
<th>Intrinsics/Builtins</th>
<th>CodeGen</th>
<th>Auto-Vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Encoding, Format, Effects,...)</td>
<td>(LLVM-IR, C/C++)</td>
<td>(ISel Patterns, Legalization)</td>
<td>(SIMD, Heuristics,...)</td>
</tr>
<tr>
<td>Extensible Compiler [DLR]</td>
<td>☑</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>(Needs user inputs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- [TUDA]</td>
<td>☑</td>
<td>☑</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>OpenASIP 2.0 [TUNI]</td>
<td>☑</td>
<td>☑</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Ours [Seal5]</td>
<td>☑</td>
<td>☑</td>
<td>(Semi-automated)</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>(Experimental)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Narrow 32-bit SIMD only)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Usage by SW Developer:

```c
asm("mac x3, x4, x5");
_builtin_mac(acc, x, y);
acc += x * y;
for (i = 0; i < n; i++) {
    acc += arr_x[i] * arr_y[i];
}
```
Why do we need patterns?

- During compilation the original program is lowered to intermediate representations (IRs) in a step-by-step fashion.
- Optimizations are applied along the way.
- During Instruction Selection, Generic LLVM instructions are converted to target-specific MachineInstructions.
- Instruction Selection depends on manually specified patterns to insert any instructions.
Generating ISel Patterns

Method
1. Convert CoreDSL behavior to LLVM-IR functions
2. Perform lowering in a similar way to target SW
3. Add hook to emit final DAG right before Instruction Selection would take place
4. Transform DAG nodes to TableGen code for patterns

Advantages
- Re-use existing code in LLVM
- Same optimizations → increased likeliness that extracted patterns will actually match
- SIMD-instructions are detected automatically
Seal5 Evaluation (Core-V)

Core-V Extension (OpenHW Group)
- 300+ ALU/Mem/SIMD/… instructions
- Implemented in [CV32E40P]

Configurations
1. Baseline (RV32IM)
2. Core-V Reference
3. Seal5 Generated
   a) Without SIMD
   b) With SIMD

Benchmarks
- 100+ embedded programs

Fig. 3. Histogram of the reduction of all benchmark program’s runtime in cycles measured on the CV32E40P core. Baseline is the runtime of the program compiled without any Core-V extension support. Δ Runtime smaller 0% indicates that the LLVM with Core-V instruction extension could improve runtime compared to a program with just standard RISC-V instructions. Programs are grouped by benchmark (colors) into runtime bins.
## Overview of Challenges

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Missing Hardware, Specifications,…</td>
<td>Virtual Prototyping (ETISS &amp; CoreDSL)</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparison between different Frameworks, Targets,… difficult</td>
<td>TinyML Deployment &amp; Benchmarking Flow (MLonMCU)</td>
</tr>
<tr>
<td>Profiling</td>
<td>Where to apply optimizations? How to find bottlenecks?</td>
<td>Multi-level profiling based on static/dynamic binary/trace analysis</td>
</tr>
<tr>
<td>Validation</td>
<td>How to guarantee Model accuracy is maintained during deployment?</td>
<td>Target-independent Validation Flow in MLonMCU (MLIFIO)</td>
</tr>
<tr>
<td>Tuning</td>
<td>Can we optimize the generated Kernels for specific targets efficiently?</td>
<td>TVM’s MetaScheduler with improvements</td>
</tr>
<tr>
<td>Retargeting</td>
<td>How to avoid manual efforts to support new target hardware?</td>
<td>Metamodel-based Generation of LLVM and TVM Patches</td>
</tr>
</tbody>
</table>

---

Philipp van Kempen (TUM) | 23.04.2024
Further challenges

Not covered in this talk:

- General Constraints of Embedded Systems
- ML Specifics
  - Model Design & Model Compression
- HW Specifics
  - RTL-generation for ASIPs (HW/SW-Codesign)
- Real-world problems
  - Flash time bottleneck & wearing out hardware

Needs to be considered at all design stages

Starting with pre-trained and quantized Models

Required to estimate implementation overhead of optimizations
Conclusion

Summary
• Without overcoming aforementioned challenges the HW/SW co-exploration (DSE) will be infeasible
• Retargeting is essential to eliminate manual efforts

Seal5 – Retargeting LLVM Compiler for RISC-V
• Novel approach for robust pattern generation and SIMD support
• Compared with reference Core-V vendor toolchain

Next steps:
• Retargeting support for ML Compilation
• Solve remaining challenges
References


[MLIF] Repo: https://github.com/tum-eda/mlonmcu-sw


[Seal5] Paper: N/A Repo: Release in May 2024