MLonMCU: TinyML Benchmarking with Fast Retargeting

Philipp van Kempen, Rafael Stahl, Daniel Mueller-Gritschneder, Ulf Schlichtmann
Chair of Electronic Design Automation, Technical University of Munich
Munich, Germany
{philipp.van-kempen, r.stahl, daniel.mueller, ulf.schlichtmann}@tum.de

13. October 2022

This work has been developed in the ZuSE project Scale4Edge. Scale4Edge is funded by the German ministry of education and research (BMBF) (reference numbers: 16ME0122K-16ME0140). The authors are responsible for the content of this publication.

SPONSORED BY THE
Federal Ministry of Education and Research

About myself

- M.Sc. in Electrical and Computer Engineering
- Doctoral Candidate at “Embedded System Level (ESL)” group since 2021

- **Focus:** Embedded Machine Learning (TinyML)
  - Deployment
  - Optimizations
  - RISC-V ISA
  - Automation

Content

1. Introduction
- Motivation
- State of the Art

2. Implementation
- Design Principles
- Components
- Demo

3. Experiments
- Methodology
- Runtime overhead of TinyML backends
- TVM schedules on MCU hardware
- Results

4. Conclusion
- Summary
- Outlook

Content

1. Introduction
- Motivation
- State of the Art

2. Implementation
- Design Principles
- Components
- Demo

3. Experiments
- Methodology
- Runtime overhead of TinyML backends
- TVM schedules on MCU hardware
- Results

4. Conclusion
- Summary
- Outlook

Motivation

TinyML: Bring machine learning tasks to tiny edge devices

- Resource constraints (memory, power, cost, size, latency)
- Sometimes connected to cloud (IoT)
- Here: inference only

Further challenges:

- Deployment: Used frameworks etc.
- Hardware Design: Reduce time-to-market (i.e. by virtual prototyping)
- Benchmarking: often framework/vendor-specific

→ **MLonMCU:** Framework-independent deployment and benchmarking tool with fast retargeting support

State of the Art

TinyML Frameworks

- **TFLM:** TensorFlow Lite for Microcontrollers [1]
  - Industry standard, hardcoded NN kernels, supports vendor libraries
  - TFLite Micro Compiler: Static code generator for TFLM inference [2]

- **TVM:** An Automated End-to-End Optimizing Compiler for Deep Learning [3]
  - Compiler-driven approach, Generated kernels, support Tuning
  - MicroTVM: Deployment platform for baremetal devices (MCUs)

Content

1. Introduction
- Motivation
- State of the Art

2. Implementation
- Design Principles
- Components
- Demo

3. Experiments
- Methodology
- Runtime overhead of TinyML backends
- TVM schedules on MCU hardware
- Results

4. Conclusion
- Summary
- Outlook

Implementation: Design Principles

Isolation

- Do not interfere with rest of the system

Reproducibility

- All intermediate benchmarking artifacts should be accessible

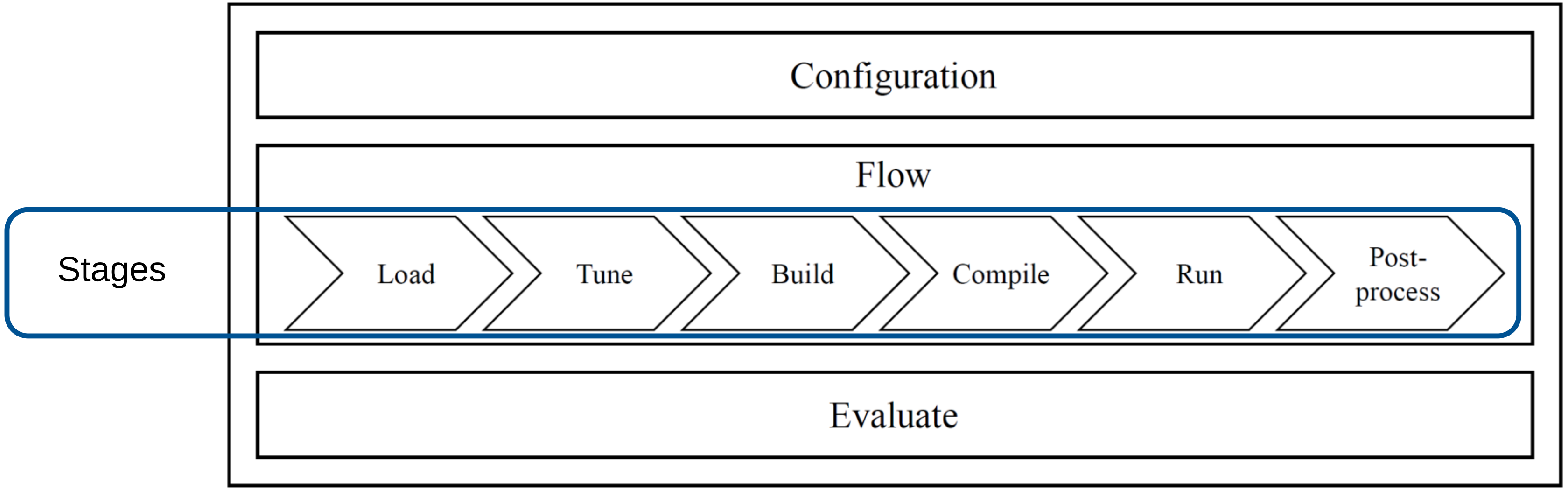
Parallelism

- Use all available computational resources

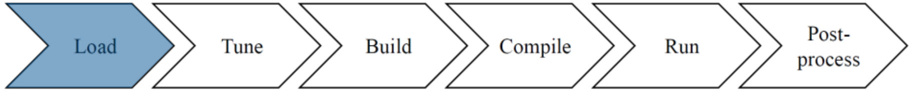
Extensibility

- Integrate custom user-written code easily

Implementation: Structure

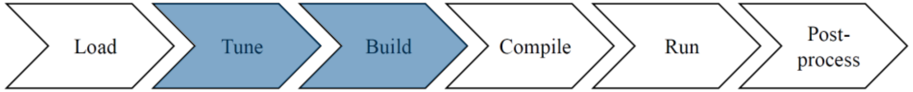


Implementation: Components I



Frontends

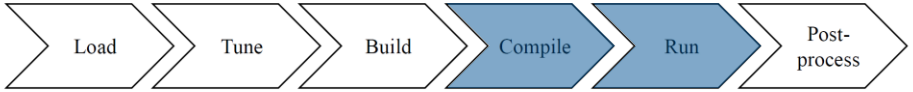
- Load: Make model (provided by name or path) available for later stages



Frameworks and Backends

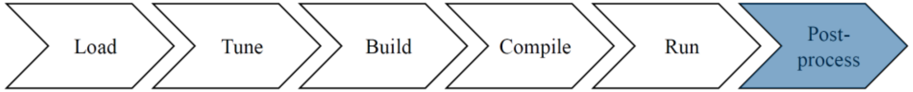
- Tune: Benchmark several different operator configurations on target hardware to reduce latency (optional)
- Build: Use provided model (and tuning records if available) for generating the models inference code and kernels

Implementation: Components II



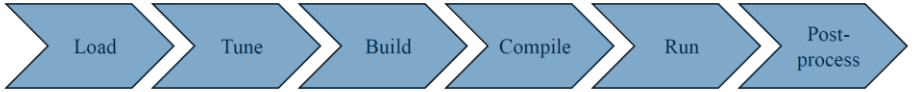
Platforms and Targets

- Compile: Produce target software binary (ELF)
- Run: Execute & monitor target software on a device or simulator



Postprocesses

- Use previously generated results (metrics & artifacts) to do further analysis and some filtering



Features

- Alter the behavior of specific components (by adding callbacks or modifying their configuration)

Demo

Used commands

- **Experiment A**

```
mlonmcu flow run tinymlperf                                                                   \
           --parallel --progress                                                              \
           --target etiss_pulpino                                                             \
           --backend tflmi --backend tflmc --backend tvmaot --backend tvmaotplus --backend tvmrt  \
           --feature trace --postprocess detailed_cycles
```

- **Experiment B**

```
mlonmcu flow run tinymlperf --parallel --progress --backend tvmaot                            \
           --target esp32c3 --target stm32f4 --target stm32f7 –target esp32                   \
           --feature-gen _ --feature-gen autotuned --config autotuned.tuning_records=...      \
           --config-gen tvmaot.desired_layout=NHWC --config-gen tvmaot.desired_layout=NCHW    \
           --config-gen tvmaot.target_device=arm_cpu tvmaot.desired_layout=NHWC               \
           --config-gen tvmaot.target_device=arm_cpu tvmaot.desired_layout=NCHW
```

Content

1. Introduction
- Motivation
- State of the Art

2. Implementation
- Design Principles
- Components
- Demo

3. Experiments
- Methodology
- Runtime overhead of TinyML backends
- TVM schedules on MCU hardware
- Results

4. Conclusion
- Summary
- Outlook

Methodology: Models

Used Models: **MLPerf Tiny Benchmark** [4]

Name	Use Case	Quantized Size
`aww`	Keyword Spotting	58.3 kB
`vww`	Visual Wake Words	325 kB
`resnet`	Image Classification	96.2 kB
`toycar`	Anomaly Detection	270 kB

(DS-)CNN

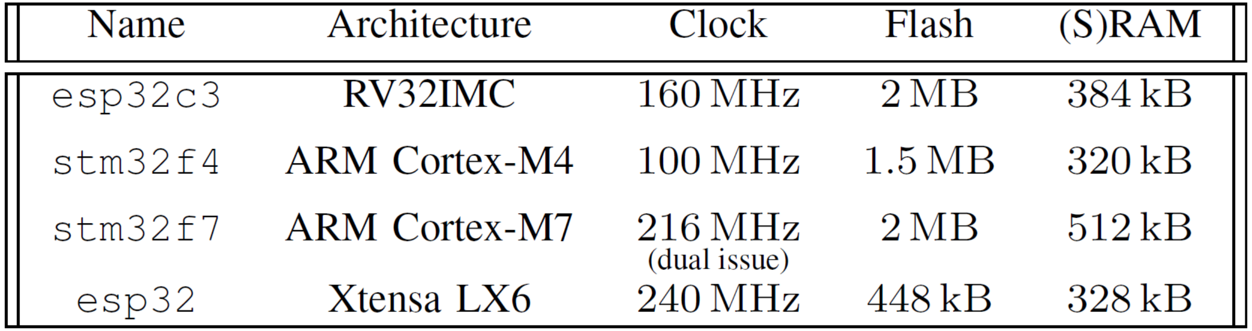
DNN

Considerations

- Quantized only

Methodology: Targets

Used Targets:



Name	Architecture	Clock	Flash	(S)RAM
`esp32c3`	RV32IMC	160 MHz	2 MB	384 kB
`stm32f4`	ARM Cortex-M4	100 MHz	1.5 MB	320 kB
`stm32f7`	ARM Cortex-M7	216 MHz (dual issue)	2 MB	512 kB
`esp32`	Xtensa LX6	240 MHz	448 kB	328 kB

Considerations

- Single core inference only
- Usefulness of these boards should not be the point of discussion
- Most limiting factor: available (S)RAM (for intermediate tensors and MicroTVM runtime)

Experiment A

Tasks

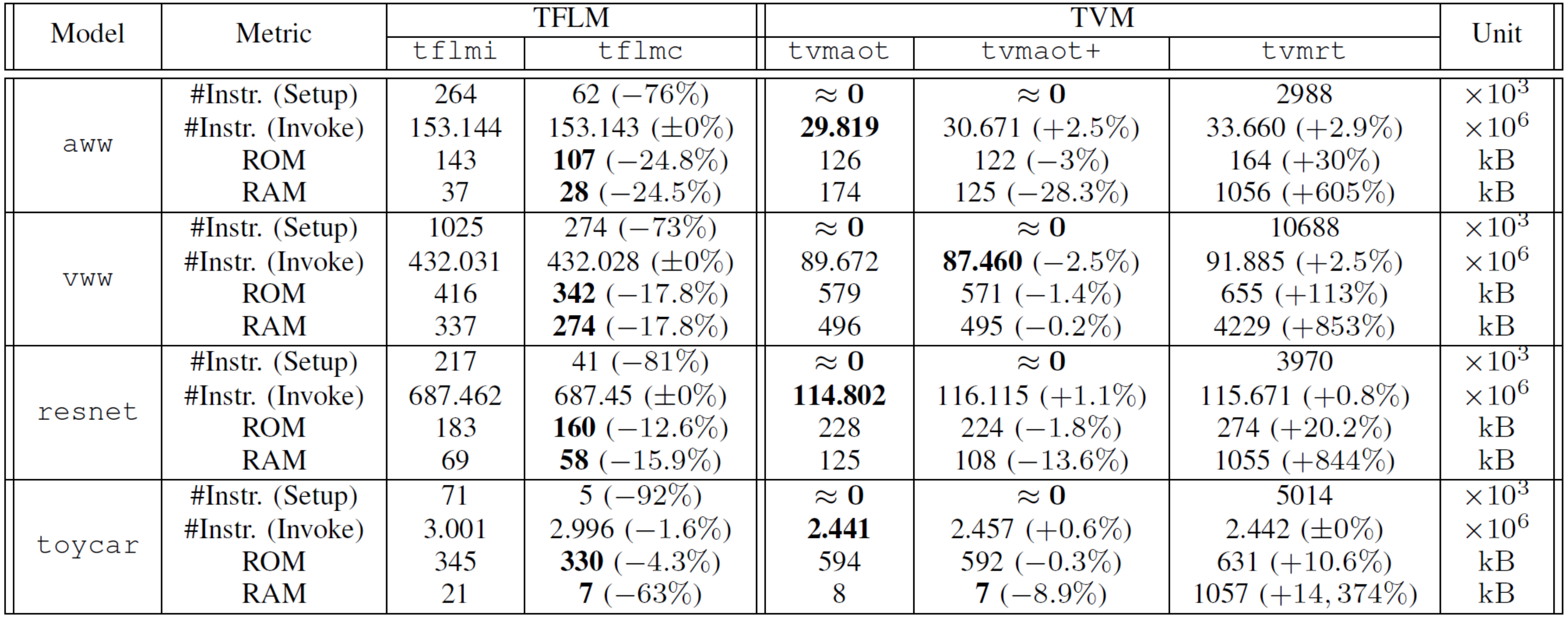
1. Compare runtime overheads of two TFLM backends
2. Compare runtime overheads of three TVM backends
3. Compare TFLM with TVM framework

Considered metrics

- Cycles (Setup & Invoke)
- Memory
  - ROM: Constant weights, code size
  - RAM: Intermediate tensors (activations), temporary workspace buffers

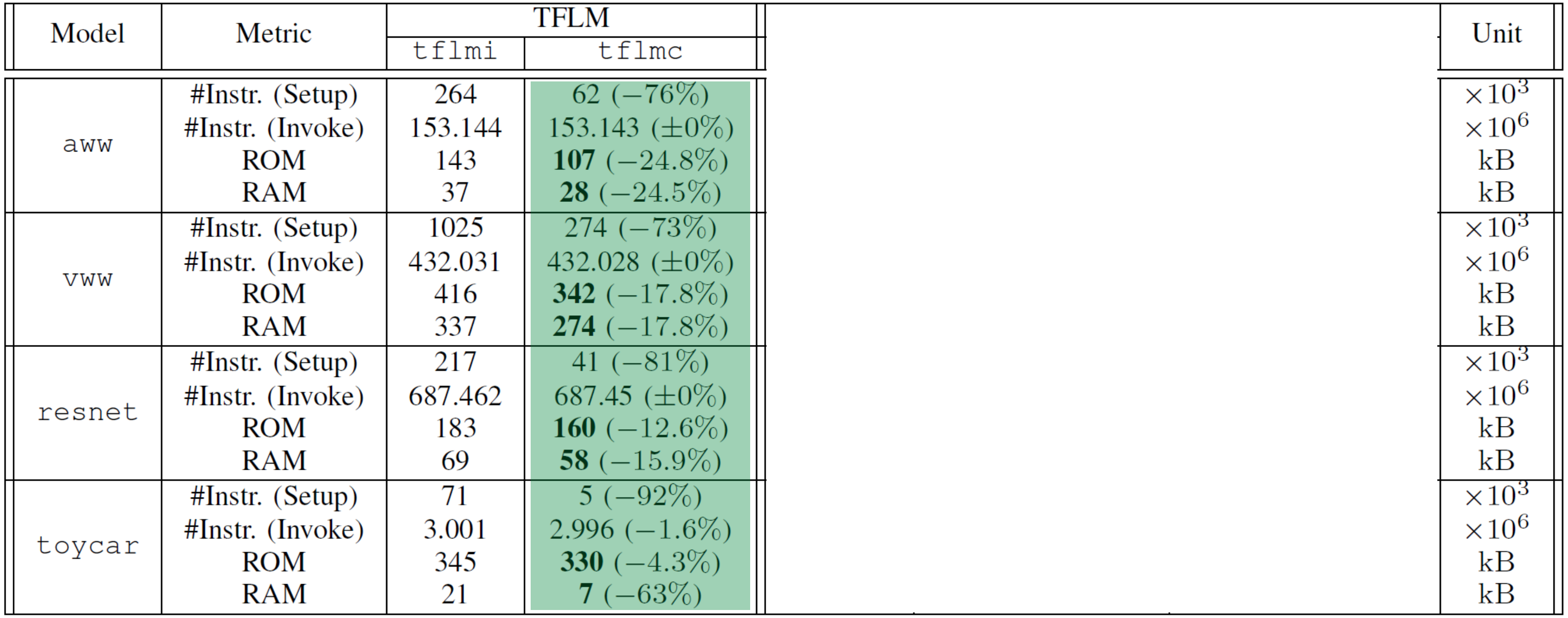
Used Simulator: ETISS (Extendable Translating Instruction Set Simulator) [5]

Experiment A: Framework/Backend Overheads



Model	Metric	TFLM		TVM			Unit
		`tflmi`	`tflmc`	`tvmaot`	`tvmaot+`	`tvmrt`	
`aww`	#Instr. (Setup)	264	62 (−76%)	≈ **0**	≈ **0**	2988	$\times 10^3$
	#Instr. (Invoke)	153.144	153.143 (±0%)	**29.819**	30.671 (+2.5%)	33.660 (+2.9%)	$\times 10^6$
	ROM	143	**107** (−24.8%)	126	122 (−3%)	164 (+30%)	kB
	RAM	37	**28** (−24.5%)	174	125 (−28.3%)	1056 (+605%)	kB
`vww`	#Instr. (Setup)	1025	274 (−73%)	≈ **0**	≈ **0**	10688	$\times 10^3$
	#Instr. (Invoke)	432.031	432.028 (±0%)	89.672	**87.460** (−2.5%)	91.885 (+2.5%)	$\times 10^6$
	ROM	416	**342** (−17.8%)	579	571 (−1.4%)	655 (+113%)	kB
	RAM	337	**274** (−17.8%)	496	495 (−0.2%)	4229 (+853%)	kB
`resnet`	#Instr. (Setup)	217	41 (−81%)	≈ **0**	≈ **0**	3970	$\times 10^3$
	#Instr. (Invoke)	687.462	687.45 (±0%)	**114.802**	116.115 (+1.1%)	115.671 (+0.8%)	$\times 10^6$
	ROM	183	**160** (−12.6%)	228	224 (−1.8%)	274 (+20.2%)	kB
	RAM	69	**58** (−15.9%)	125	108 (−13.6%)	1055 (+844%)	kB
`toycar`	#Instr. (Setup)	71	5 (−92%)	≈ **0**	≈ **0**	5014	$\times 10^3$
	#Instr. (Invoke)	3.001	2.996 (−1.6%)	**2.441**	2.457 (+0.6%)	2.442 (±0%)	$\times 10^6$
	ROM	345	**330** (−4.3%)	594	592 (−0.3%)	631 (+10.6%)	kB
	RAM	21	**7** (−63%)	8	**7** (−8.9%)	1057 (+14, 374%)	kB

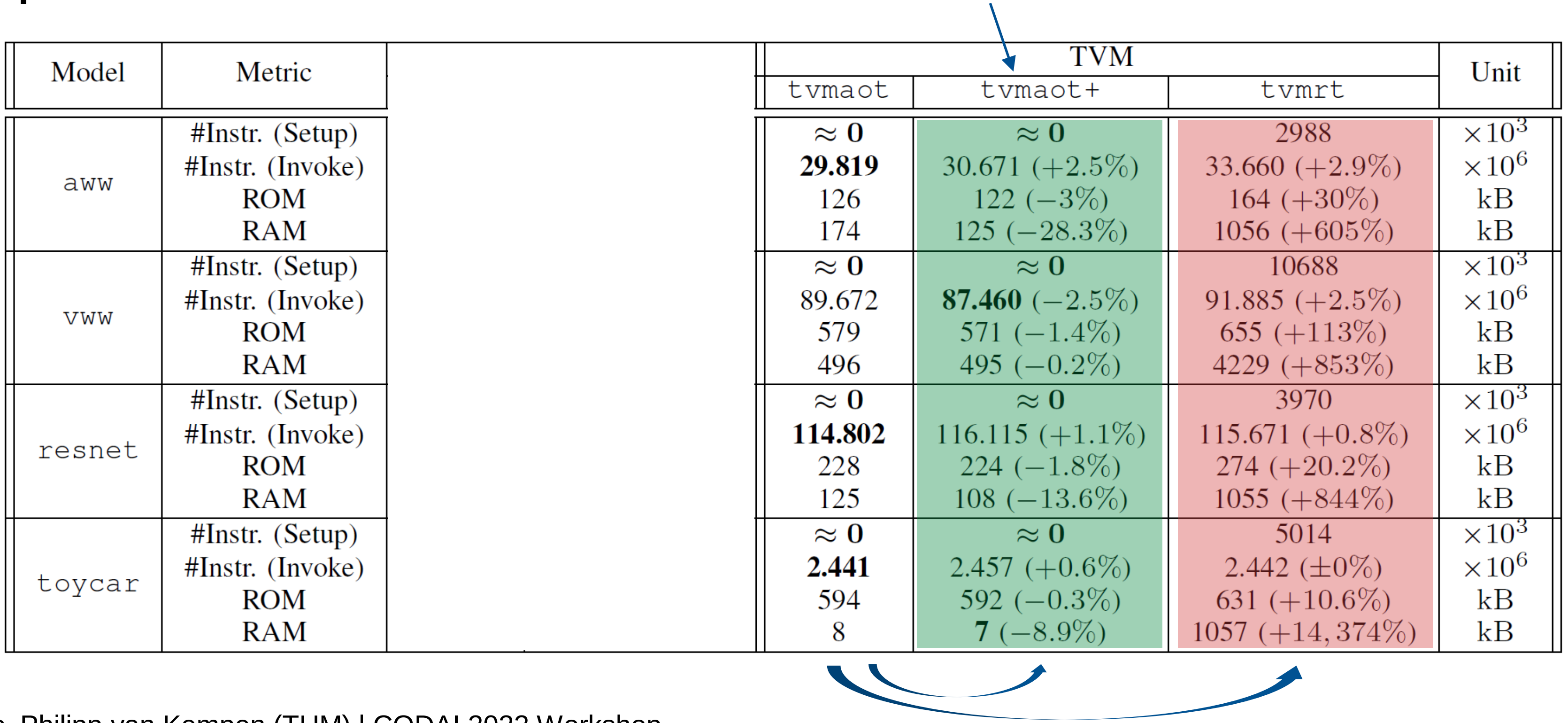
Experiment A: TFLM Backends



Model	Metric	TFLM		Unit
		`tflmi`	`tflmc`	
`aww`	#Instr. (Setup)	264	62 (−76%)	$\times10^3$
	#Instr. (Invoke)	153.144	153.143 (±0%)	$\times10^6$
	ROM	143	**107** (−24.8%)	kB
	RAM	37	**28** (−24.5%)	kB
`vww`	#Instr. (Setup)	1025	274 (−73%)	$\times10^3$
	#Instr. (Invoke)	432.031	432.028 (±0%)	$\times10^6$
	ROM	416	**342** (−17.8%)	kB
	RAM	337	**274** (−17.8%)	kB
`resnet`	#Instr. (Setup)	217	41 (−81%)	$\times10^3$
	#Instr. (Invoke)	687.462	687.45 (±0%)	$\times10^6$
	ROM	183	**160** (−12.6%)	kB
	RAM	69	**58** (−15.9%)	kB
`toycar`	#Instr. (Setup)	71	5 (−92%)	$\times10^3$
	#Instr. (Invoke)	3.001	2.996 (−1.6%)	$\times10^6$
	ROM	345	**330** (−4.3%)	kB
	RAM	21	**7** (−63%)	kB

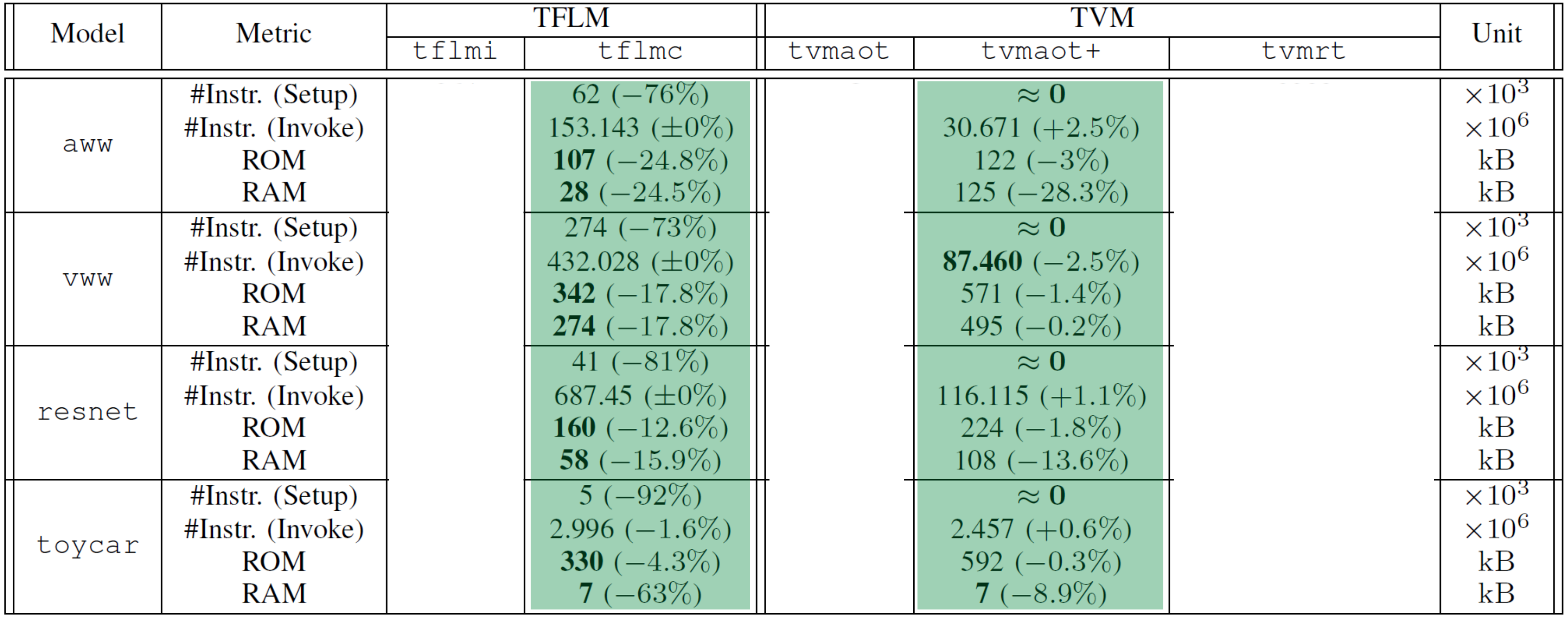
Experiment A: TVM Backends

`tvmaot+ := tvmaot + usmp + unpacked_api`



Model	Metric	TVM			Unit
		`tvmaot`	`tvmaot+`	`tvmrt`	
`aww`	#Instr. (Setup)	≈ 0	≈ 0	2988	$\times 10^3$
	#Instr. (Invoke)	**29.819**	30.671 (+2.5%)	33.660 (+2.9%)	$\times 10^6$
	ROM	126	122 (−3%)	164 (+30%)	kB
	RAM	174	125 (−28.3%)	1056 (+605%)	kB
`vww`	#Instr. (Setup)	≈ 0	≈ 0	10688	$\times 10^3$
	#Instr. (Invoke)	89.672	**87.460** (−2.5%)	91.885 (+2.5%)	$\times 10^6$
	ROM	579	571 (−1.4%)	655 (+113%)	kB
	RAM	496	495 (−0.2%)	4229 (+853%)	kB
`resnet`	#Instr. (Setup)	≈ 0	≈ 0	3970	$\times 10^3$
	#Instr. (Invoke)	**114.802**	116.115 (+1.1%)	115.671 (+0.8%)	$\times 10^6$
	ROM	228	224 (−1.8%)	274 (+20.2%)	kB
	RAM	125	108 (−13.6%)	1055 (+844%)	kB
`toycar`	#Instr. (Setup)	≈ 0	≈ 0	5014	$\times 10^3$
	#Instr. (Invoke)	**2.441**	2.457 (+0.6%)	2.442 (±0%)	$\times 10^6$
	ROM	594	592 (−0.3%)	631 (+10.6%)	kB
	RAM	8	**7** (−8.9%)	1057 (+14,374%)	kB

Experiment A: TFLM vs. TVM Framework



Model	Metric	TFLM		TVM			Unit
		`tflmi`	`tflmc`	`tvmaot`	`tvmaot+`	`tvmrt`	
`aww`	#Instr. (Setup)		62 (−76%)		**≈ 0**		$\times 10^3$
	#Instr. (Invoke)		153.143 (±0%)		30.671 (+2.5%)		$\times 10^6$
	ROM		**107** (−24.8%)		122 (−3%)		kB
	RAM		**28** (−24.5%)		125 (−28.3%)		kB
`vww`	#Instr. (Setup)		274 (−73%)		**≈ 0**		$\times 10^3$
	#Instr. (Invoke)		432.028 (±0%)		**87.460** (−2.5%)		$\times 10^6$
	ROM		**342** (−17.8%)		571 (−1.4%)		kB
	RAM		**274** (−17.8%)		495 (−0.2%)		kB
`resnet`	#Instr. (Setup)		41 (−81%)		**≈ 0**		$\times 10^3$
	#Instr. (Invoke)		687.45 (±0%)		116.115 (+1.1%)		$\times 10^6$
	ROM		**160** (−12.6%)		224 (−1.8%)		kB
	RAM		**58** (−15.9%)		108 (−13.6%)		kB
`toycar`	#Instr. (Setup)		5 (−92%)		**≈ 0**		$\times 10^3$
	#Instr. (Invoke)		2.996 (−1.6%)		2.457 (+0.6%)		$\times 10^6$
	ROM		**330** (−4.3%)		592 (−0.3%)		kB
	RAM		**7** (−63%)		**7** (−8.9%)		kB

Experiment A: Results

Observations

- Interpreter based backends (tflmi, tvmrt) are outperformed by static alternatives
- TVMs default kernel implementation are much faster compared to TFLM reference kernels
- Memory overheads compared to TFLM exist for TVM, even with USMP feature enabled

Caveats

- TVM Graph runtime (tvmrt) has useful features (Debugging, Tuning,…)
- RAM usage of TVM backends can often be reduced by a factor of two by disabling specific legalization passes

Important

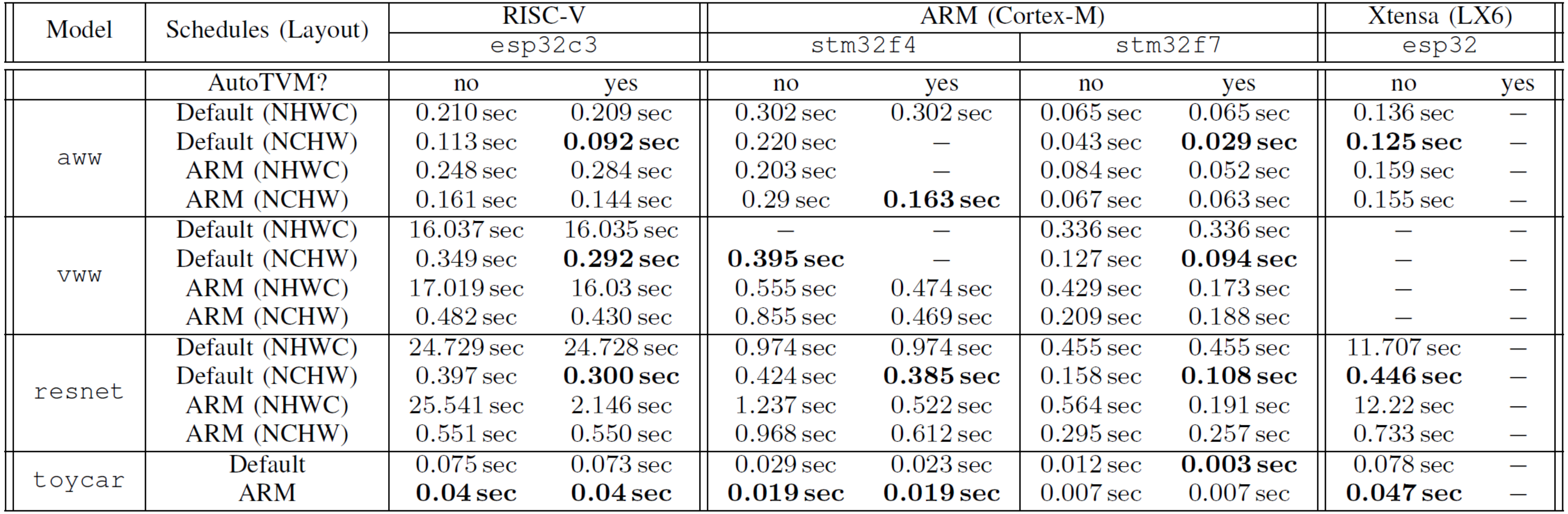
- Only considered default (unoptimized) kernel variants here
- Instruction counts instead of cycles counts

Experiment B

Tasks
1. Compare different data/kernel layouts (channels-first vs. channels-last)
2. Try out alternative TVM schedules (Default (x86) vs. ARM)
3. Observe impact of AutoTVM (tuned vs. untuned)

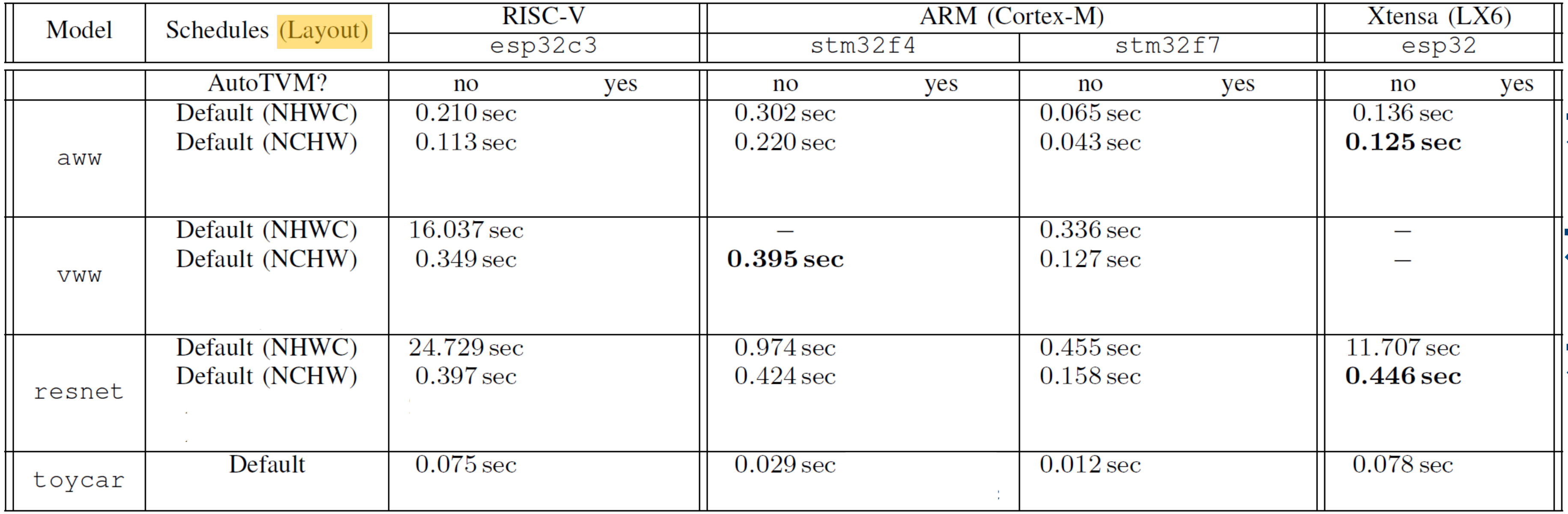
→ 4 Targets x 4 Models x 4-8 Schedules = **98 Measurements**
(excluding invalid/unsupported combinations)

Experiment B: TVM Schedules on actual HW



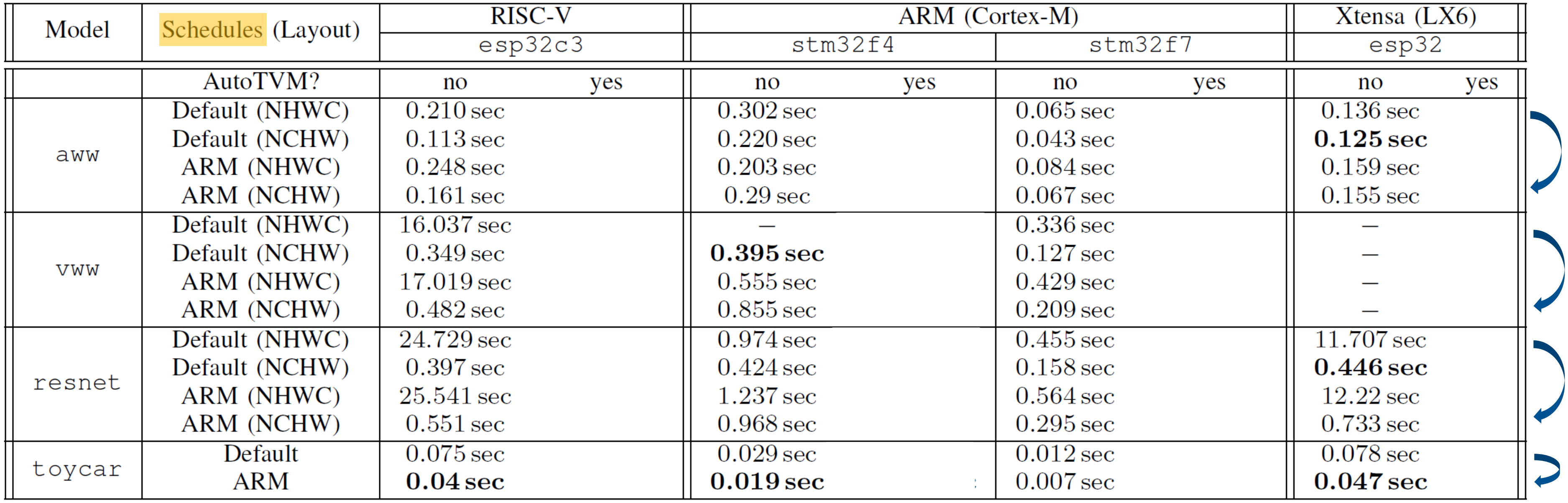
Model	Schedules (Layout)	RISC-V		ARM (Cortex-M)				Xtensa (LX6)	
		`esp32c3`		`stm32f4`		`stm32f7`		`esp32`	
	AutoTVM?	no	yes	no	yes	no	yes	no	yes
`aww`	Default (NHWC)	0.210 sec	0.209 sec	0.302 sec	0.302 sec	0.065 sec	0.065 sec	0.136 sec	–
	Default (NCHW)	0.113 sec	**0.092 sec**	0.220 sec	–	0.043 sec	**0.029 sec**	**0.125 sec**	–
	ARM (NHWC)	0.248 sec	0.284 sec	0.203 sec	–	0.084 sec	0.052 sec	0.159 sec	–
	ARM (NCHW)	0.161 sec	0.144 sec	0.29 sec	**0.163 sec**	0.067 sec	0.063 sec	0.155 sec	–
`vww`	Default (NHWC)	16.037 sec	16.035 sec	–	–	0.336 sec	0.336 sec	–	–
	Default (NCHW)	0.349 sec	**0.292 sec**	**0.395 sec**	–	0.127 sec	**0.094 sec**	–	–
	ARM (NHWC)	17.019 sec	16.03 sec	0.555 sec	0.474 sec	0.429 sec	0.173 sec	–	–
	ARM (NCHW)	0.482 sec	0.430 sec	0.855 sec	0.469 sec	0.209 sec	0.188 sec	–	–
`resnet`	Default (NHWC)	24.729 sec	24.728 sec	0.974 sec	0.974 sec	0.455 sec	0.455 sec	11.707 sec	–
	Default (NCHW)	0.397 sec	**0.300 sec**	0.424 sec	**0.385 sec**	0.158 sec	**0.108 sec**	**0.446 sec**	–
	ARM (NHWC)	25.541 sec	2.146 sec	1.237 sec	0.522 sec	0.564 sec	0.191 sec	12.22 sec	–
	ARM (NCHW)	0.551 sec	0.550 sec	0.968 sec	0.612 sec	0.295 sec	0.257 sec	0.733 sec	–
`toycar`	Default	0.075 sec	0.073 sec	0.029 sec	0.023 sec	0.012 sec	**0.003 sec**	0.078 sec	–
	ARM	**0.04 sec**	**0.04 sec**	**0.019 sec**	**0.019 sec**	0.007 sec	0.007 sec	**0.047 sec**	–

Experiment B: NHWC vs. NCHW Layouts



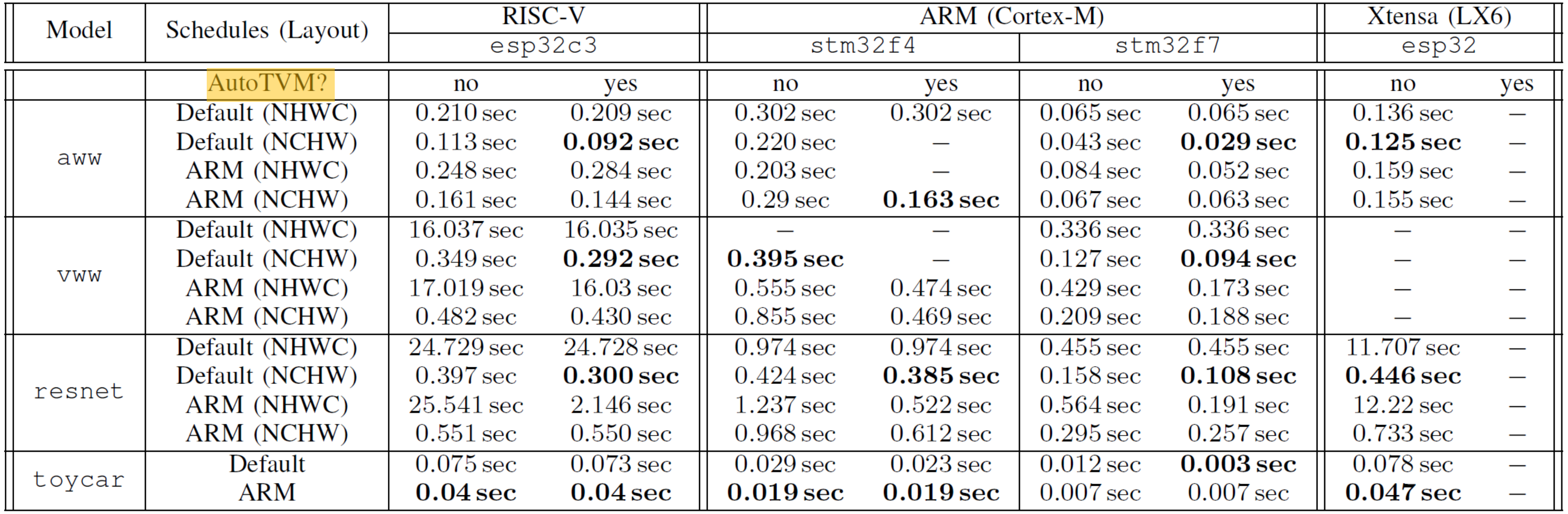
Model	Schedules (Layout)	RISC-V		ARM (Cortex-M)				Xtensa (LX6)	
		esp32c3		stm32f4		stm32f7		esp32	
	AutoTVM?	no	yes	no	yes	no	yes	no	yes
aww	Default (NHWC)	0.210 sec		0.302 sec		0.065 sec		0.136 sec	
	Default (NCHW)	0.113 sec		0.220 sec		0.043 sec		**0.125 sec**	
vww	Default (NHWC)	16.037 sec		–		0.336 sec		–	
	Default (NCHW)	0.349 sec		**0.395 sec**		0.127 sec		–	
resnet	Default (NHWC)	24.729 sec		0.974 sec		0.455 sec		11.707 sec	
	Default (NCHW)	0.397 sec		0.424 sec		0.158 sec		**0.446 sec**	
toycar	Default	0.075 sec		0.029 sec		0.012 sec		0.078 sec	

Experiment B: Default vs. ARM Schedules



Model	Schedules (Layout)	RISC-V		ARM (Cortex-M)				Xtensa (LX6)	
		esp32c3		stm32f4		stm32f7		esp32	
	AutoTVM?	no	yes	no	yes	no	yes	no	yes
aww	Default (NHWC)	0.210 sec		0.302 sec		0.065 sec		0.136 sec	
	Default (NCHW)	0.113 sec		0.220 sec		0.043 sec		**0.125 sec**	
	ARM (NHWC)	0.248 sec		0.203 sec		0.084 sec		0.159 sec	
	ARM (NCHW)	0.161 sec		0.29 sec		0.067 sec		0.155 sec	
vww	Default (NHWC)	16.037 sec		–		0.336 sec		–	
	Default (NCHW)	0.349 sec		**0.395 sec**		0.127 sec		–	
	ARM (NHWC)	17.019 sec		0.555 sec		0.429 sec		–	
	ARM (NCHW)	0.482 sec		0.855 sec		0.209 sec		–	
resnet	Default (NHWC)	24.729 sec		0.974 sec		0.455 sec		11.707 sec	
	Default (NCHW)	0.397 sec		0.424 sec		0.158 sec		**0.446 sec**	
	ARM (NHWC)	25.541 sec		1.237 sec		0.564 sec		12.22 sec	
	ARM (NCHW)	0.551 sec		0.968 sec		0.295 sec		0.733 sec	
toycar	Default	0.075 sec		0.029 sec		0.012 sec		0.078 sec	
	ARM	**0.04 sec**		**0.019 sec**		0.007 sec		**0.047 sec**	

Experiment B: Untuned vs. Tuned



Model	Schedules (Layout)	RISC-V		ARM (Cortex-M)				Xtensa (LX6)	
		esp32c3		stm32f4		stm32f7		esp32	
	AutoTVM?	no	yes	no	yes	no	yes	no	yes
aww	Default (NHWC)	0.210 sec	0.209 sec	0.302 sec	0.302 sec	0.065 sec	0.065 sec	0.136 sec	–
	Default (NCHW)	0.113 sec	**0.092 sec**	0.220 sec	–	0.043 sec	**0.029 sec**	**0.125 sec**	–
	ARM (NHWC)	0.248 sec	0.284 sec	0.203 sec	–	0.084 sec	0.052 sec	0.159 sec	–
	ARM (NCHW)	0.161 sec	0.144 sec	0.29 sec	**0.163 sec**	0.067 sec	0.063 sec	0.155 sec	–
vww	Default (NHWC)	16.037 sec	16.035 sec	–	–	0.336 sec	0.336 sec	–	–
	Default (NCHW)	0.349 sec	**0.292 sec**	**0.395 sec**	–	0.127 sec	**0.094 sec**	–	–
	ARM (NHWC)	17.019 sec	16.03 sec	0.555 sec	0.474 sec	0.429 sec	0.173 sec	–	–
	ARM (NCHW)	0.482 sec	0.430 sec	0.855 sec	0.469 sec	0.209 sec	0.188 sec	–	–
resnet	Default (NHWC)	24.729 sec	24.728 sec	0.974 sec	0.974 sec	0.455 sec	0.455 sec	11.707 sec	–
	Default (NCHW)	0.397 sec	**0.300 sec**	0.424 sec	**0.385 sec**	0.158 sec	**0.108 sec**	**0.446 sec**	–
	ARM (NHWC)	25.541 sec	2.146 sec	1.237 sec	0.522 sec	0.564 sec	0.191 sec	12.22 sec	–
	ARM (NCHW)	0.551 sec	0.550 sec	0.968 sec	0.612 sec	0.295 sec	0.257 sec	0.733 sec	–
toycar	Default	0.075 sec	0.073 sec	0.029 sec	0.023 sec	0.012 sec	**0.003 sec**	0.078 sec	–
	ARM	**0.04 sec**	**0.04 sec**	**0.019 sec**	**0.019 sec**	0.007 sec	0.007 sec	**0.047 sec**	–

Experiment B: Untuned vs. Tuned

rel. Improvement:

🟥	🟨	🟩
0-5 %	5-20 %	> 20 %

	Model	Schedules (Layout)	RISC-V		ARM (Cortex-M)				Xtensa (LX6)	
			esp32c3		stm32f4		stm32f7		esp32	
		AutoTVM?	no	yes	no	yes	no	yes	no	yes
	aww	Default (NHWC) Default (NCHW) ARM (NHWC) ARM (NCHW)								
	vww	Default (NHWC) Default (NCHW) ARM (NHWC) ARM (NCHW)								
CNN	resnet	Default (NHWC)	24.729 sec	24.728 sec	0.974 sec	0.974 sec	0.455 sec	0.455 sec		
		Default (NCHW)	0.397 sec	**0.300 sec**	0.424 sec	**0.385 sec**	0.158 sec	**0.108 sec**		
		ARM (NHWC)	25.541 sec	2.146 sec	1.237 sec	0.522 sec	0.564 sec	0.191 sec		
		ARM (NCHW)	0.551 sec	0.550 sec	0.968 sec	0.612 sec	0.295 sec	0.257 sec		
DNN	toycar	Default	0.075 sec	0.073 sec	0.029 sec	0.023 sec	0.012 sec	**0.003 sec**		
		ARM	**0.04 sec**	**0.04 sec**	**0.019 sec**	**0.019 sec**	0.007 sec	0.007 sec		

Experiment B: Results

Observations

- CNNs: Transform layouts to **NCHW** and **tune** kernels
- DNNs: Prefer to use untuned ARM schedules **or** tune default (x86) ones

Caveats

- Exceptions exist
- Tuning is very time/resource intensive (Especially using MicroTVM)

Important

- Optimized ARM Cortex-M (DSP) schedules NOT used here
- Optimized RISC-V Schedules for TVM are not yet available

Content

1. Introduction

- Motivation
- State of the Art

2. Implementation

- Design Principles
- Components
- Demo

3. Experiments

- Methodology
- Runtime overhead of TinyML backends
- TVM schedules on MCU hardware
- Results

4. Conclusion

- Summary
- Outlook

Overview

Achievements

- Proposed and implemented MLonMCU software with retargeting possibilities
- Many supported Components (Frontends, Frameworks, Targets,…)
- Demonstrated how to use MLonMCU effortlessly to generate complex benchmarks
- Discussed overserved results

Results

- 118 Benchmarks generated in less than one hour
- There is no “catch-all” solution for TinyML deployment
- Static code-generation outperforms interpreter-based approaches
- TFLM: Performance with “Reference” (Default) kernels miserable → Highly relies on vendor libraries
- TVM: Used layouts and Auto-Tuning highly relevant, tweak passes to optimize RAM usage on MCUs

Outlook

Work In Progress

- Improve Benchmarking Speed (Remote Execution etc.)
- RTL Simulation (Pulp Platform Cores)

Future Research

- ISA Extensions (i.e. RISC-V **P**acked & **V**ector)
- Kernel Libraries (i.e. CMSIS-NN[6])
- Network Architecture Search
- Power Consumption

Any contributions are welcome!

Try out MLonMCU by yourself!

Open Source Repositories

- **MLonMCU Package: https://github.com/tum-ei-eda/mlonmcu**
- MLonMCU Model Zoo: https://github.com/tum-ei-eda/mlonmcu-models
- MLonMCU SW Library (Machine Learning Interface): https://github.com/tum-ei-eda/mlonmcu-sw

Documentation: https://mlonmcu.readthedocs.io

Demo: https://github.com/tum-ei-eda/mlonmcu/blob/main/ipynb/Demo.ipynb

Available via PyPI: https://pypi.org/project/mlonmcu

References

[1] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang et al., “Tensorflow lite micro: Embedded machine learning for tinyml systems,” Proceedings of Machine Learning and Systems, vol. 3, pp. 800–811, 2021.
[2] R. Stahl, “exploring static code generation and simd-acceleration for machine learning on risc-v” in in “risc-v forum: Developer tools & toolchains”, 2021
[3] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze et al., “{TVM}: An automated {End-to-End} optimizing compiler for deep learning,” in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 578–594.
[4] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau et al., “Mlperf tiny benchmark,” arXiv preprint arXiv:2106.07597, 2021.
[5] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, “The extendable translating instruction set simulator (etiss) interlinked with an mda framework for fast risc prototyping,” in 2017 International Symposium on Rapid System Prototyping (RSP). IEEE, 2017, pp. 79–84.
[6] L. Lai, N. Suda, and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” arXiv preprint arXiv:1801.06601, 2018.

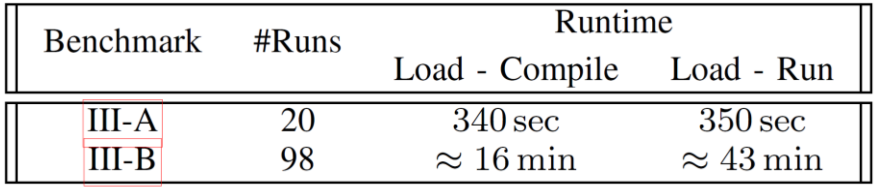
End

Bonus: Further details

- MLonMCU features…
  - High test coverage
  - Extensive Documentation
  - CI/CD Infrastructure
  - …

Bonus: Benchmark Runtimes

- Used HW: Intel(R) Core(TM) i7-6700 CPU @ 3.4GHz



Benchmark	#Runs	Runtime	
		Load - Compile	Load - Run
III-A	20	340 sec	350 sec
III-B	98	$\approx$ 16 min	$\approx$ 43 min

Bonus: MLonMCU Components

- Frontends: **TFLite**, TF, ONNX, (Paddle)
- Frameworks: **TFLM + TVM**
- Backends: TFLM Intepreter, TFLM Compiler, TVM AoT, TVM Graph, …
- Platforms/Targets:
  - Default (MLF): x86 (Host), **ETISS, Spike, OVPSim, RISC-V QEMU, Corstone300**
  - Zephyr: STM32 Discovery Boards, ESP32(-C3) and probably many more
  - ESP-IDF: ESP32-(C3) etc.
  - MicroTVM: Spike, ETISS & various Zephyr/Arduino boards → Used for AutoTVM
- Postprocesses: …
- Features:
  - muRISCV-NN (TFLM+TVM), **CMSIS-NN** (TFLM+TVM), **RISC-V Extensions (V+P)**, ARM-DSP, ARM-MVEI, **Debug**, GDBServer, Trace (Memory+Intructions), Unpacked API, **USMP**, MOIOPT, Visualize (TFLite+Relay), **Autotune**, Cache Simulation, Benchmark, RPC, Profile…