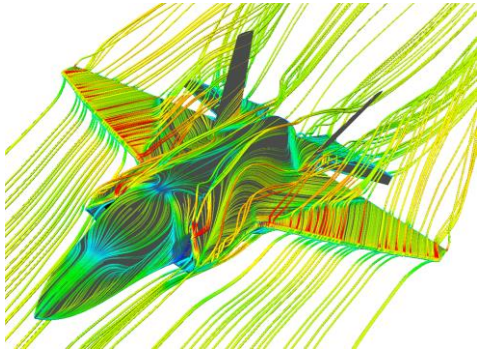
BACKPORTING RISC-V VECTOR ASSEMBLY

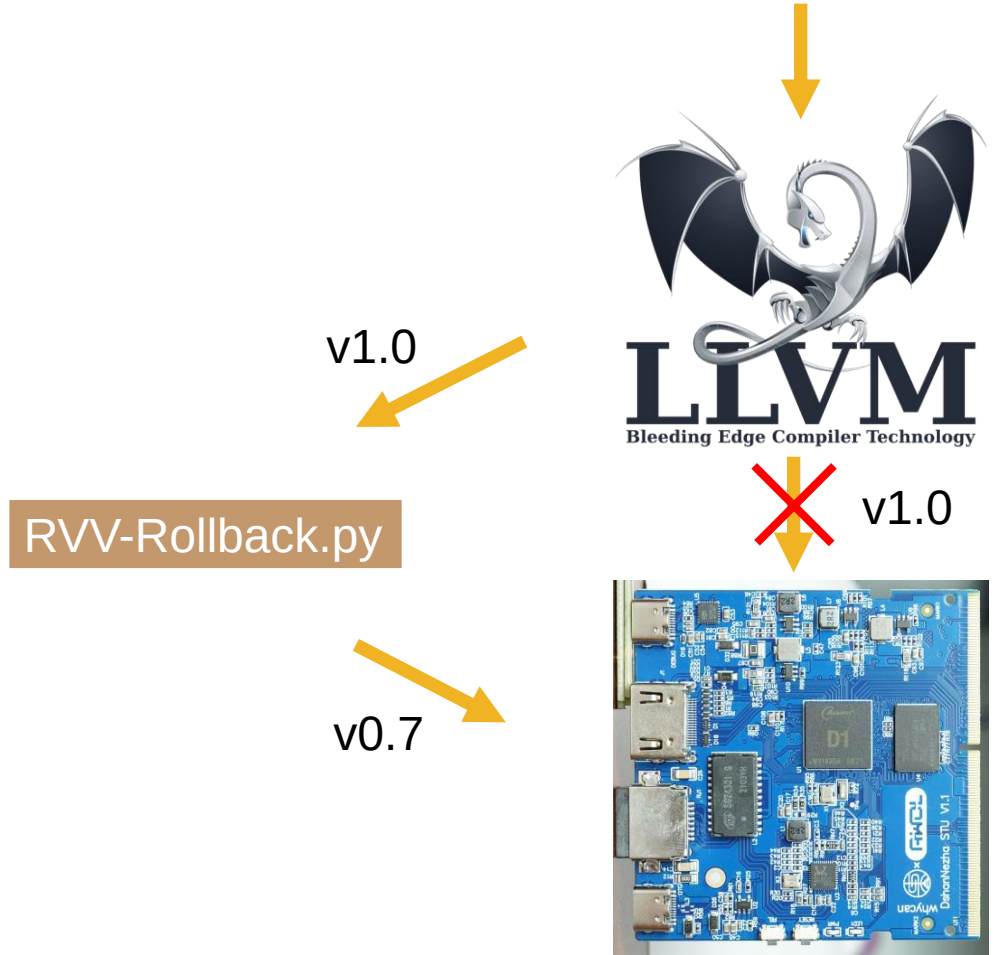
Joseph K. L. Lee, Maurice Jamieson, Nick Brown

epcc

Outline

- RISC-V Vector Extension (RVV): Brief history and Versions
- RVV-rollback tool: v1.0 to v0.7
- Vector Benchmark: Clang
- Summary





RISC-V Vector Extension (RVV): brief history

- Key feature of RISC-V: modular extensions

- 2015: Vector extension first proposed
  - Key ideas: Cray-style, variable sized vectors
  - Reconfigure element size and vector length at run time
  - Flexible for different microarchitecture implementation
  - Small set of instructions (vs typical packed SIMD alternatives)

- 2019: Major version 0.7
  - *“intended to be stable enough to begin developing toolchains, functional simulators, and initial implementations”*
  - warning that “*backwards-incompatible changes will be made prior to ratification”*
  - hardware implementations were developed

- 2021: Ratified at version 1.0

- Current and future work:
  - Intrinsic API

RISC-V Vector Extension (RVV): v0.7 vs v1.0

- 32 vector registers
- Implementation defined by:
  - ELEN: Max vector element size in bits
  - VLEN: Single vector register size in bits
- Runtime settings:
  - SEW: Selected element width
  - VL: Operational vector length (number of elements to be updated from a vector instruction)
  - LMUL: Vector register multiplier (number of vector registers grouped together)

	V0.7	V1.0
Configuration-setting instructions	*vsetvl , vsetvli*	+ *vsetivli* (immediate VL value)
LMUL	1,2,4,8	+fractional ½, ¼, ⅛ (useful for mixed width operations, reduce register spilling)
Tail/mask agnostic policy	0's in tail elements Inactive elements undisturbed	Set tail and inactive undisturbed/agnostic policy explicitly in vset (ta/tu)(ma/mu)

Other changes: simplified mask register layout, new whole register instructions, renamed instructions…

Instruction encodings are different, not binary compatible!

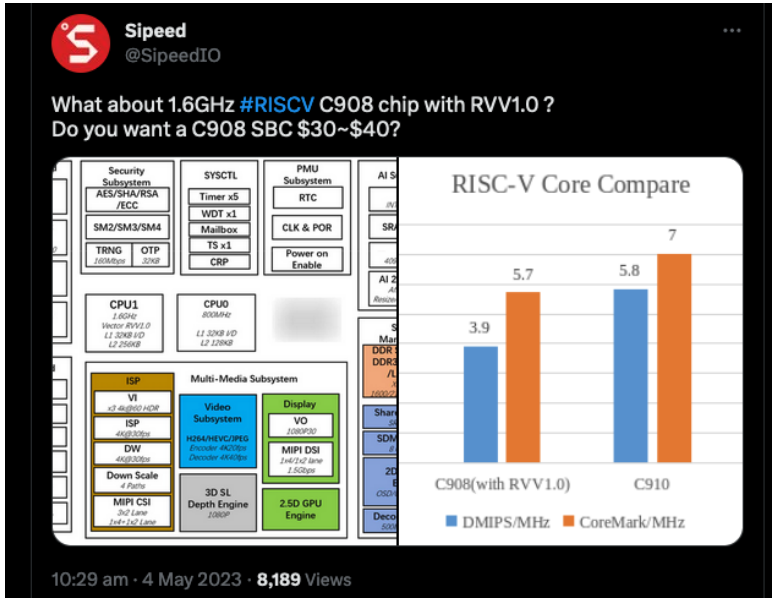
epcc

Why care about v0.7?

- Hardware: off-the-shelf only v0.7: Allwinner D1 SoC with T-Head XuanTie C906
- No v1.0 commercially available yet:
  - E.g. XuanTie C908 with RVV 1.0
- Compiler toolchains:

GNU

- Upstream GNU toolchain does not support vector extension
- rvv-next branch – limited support for RVV 1.0
- Older deleted branch rvv-0.7.1 (compiled mirror on EPCC website)
- T-Head provides modified GNU toolchain targeting C906
  - GCC 8.4 – Good auto-vectorisation (RVV 0.7)
  - GCC 10.2 – Intrinsics support, poor auto-vectorisation (RVV 0.7 & 1.0)
  - Mirror on EPCC website



LLVM

- LLVM 15 and 16 support RVV v1.0
- Support vector length agnostic (--scalable-vectorization=on) or vector length specific (--riscv-v-vector-bits-min/max=N)
- Support standard extensions with minimum vector length Zvl*, and embedded processors Zve*

RVV Rollback Tool

- We want to use Clang (and future updated compilers) on current hardware (RVV 0.7)
- One method is to emulate vector instructions (e.g. Vehave)
  - Sacrifice performance
  - We have vector hardware
- Developed a python tool: rvv-rollback
- https://github.com/RISCVtestbed/rvv-rollback
- This helps us translate RVV 1.0 assembly into RVV 0.7 assembly

RVV Rollback Tool

- Example workflow:

1. Compile with Clang 15 to obtain RVV 1.0 assembly
   - with -march=rv64gcv
   - -riscv-v-vector-bits-min=128 for VLS
   - -scalable-vectorization=on for VLA,
   - use -no-integrated-as

2. Translate RVV 1.0 assembly to RVV 0.7 using rvv-rollback.py

3. Assemble using T-Head's GNU assembler
   - v2.6.1 Xuantie-900-gcc-linux toolchain, available at https://datashare.ed.ac.uk/handle/10283/4835

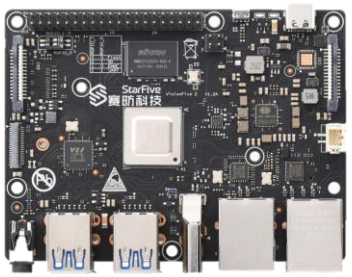
RVV Rollback Tool

- This allows us to use the optimizations and automatic vectorisations identified and applied by Clang
- Limitations:
  - Some features not supported: e.g. Fractional LMUL
  - Immediate value instructions (e.g. vsetivli) / whole register load/store (vl1r, vs1r) :
    - Default store vector setting in memory -> reconfigure -> execute -> load vector setting back
    - Adds overhead, and often unnecessary because a temporary register can be used, or redundant
    - The tool will give suggestion, and user can manually determine appropriate translation
  - For testing and benchmarking purpose, should definitely check output

Vector Benchmark: Setup

- Benchmark: RAJA Performance Suite
  - https://github.com/LLNL/RAJAPerf
- Only run single-precision floats
  - (C906 vector only supports up to 32 bits)
- Only on single core

	Allwinner D1	StarFive JH7110 (VF2)
Processor	XuanTie C906	SiFive U74
Clock Speed	1.0 GHz	1.5 GHz
Cores	1	4
Cache	32 KB I-cache + 32 KB D-cache	32 KB I-cache + 32 KB D-cache + 2MB L2
Memory	512MB DDR3	8GB DDR4
ISA	RV64GC+V0.7	RV64GC
Vector width	128 bit	N/A



epcc

Vector Benchmark: Compiler flags

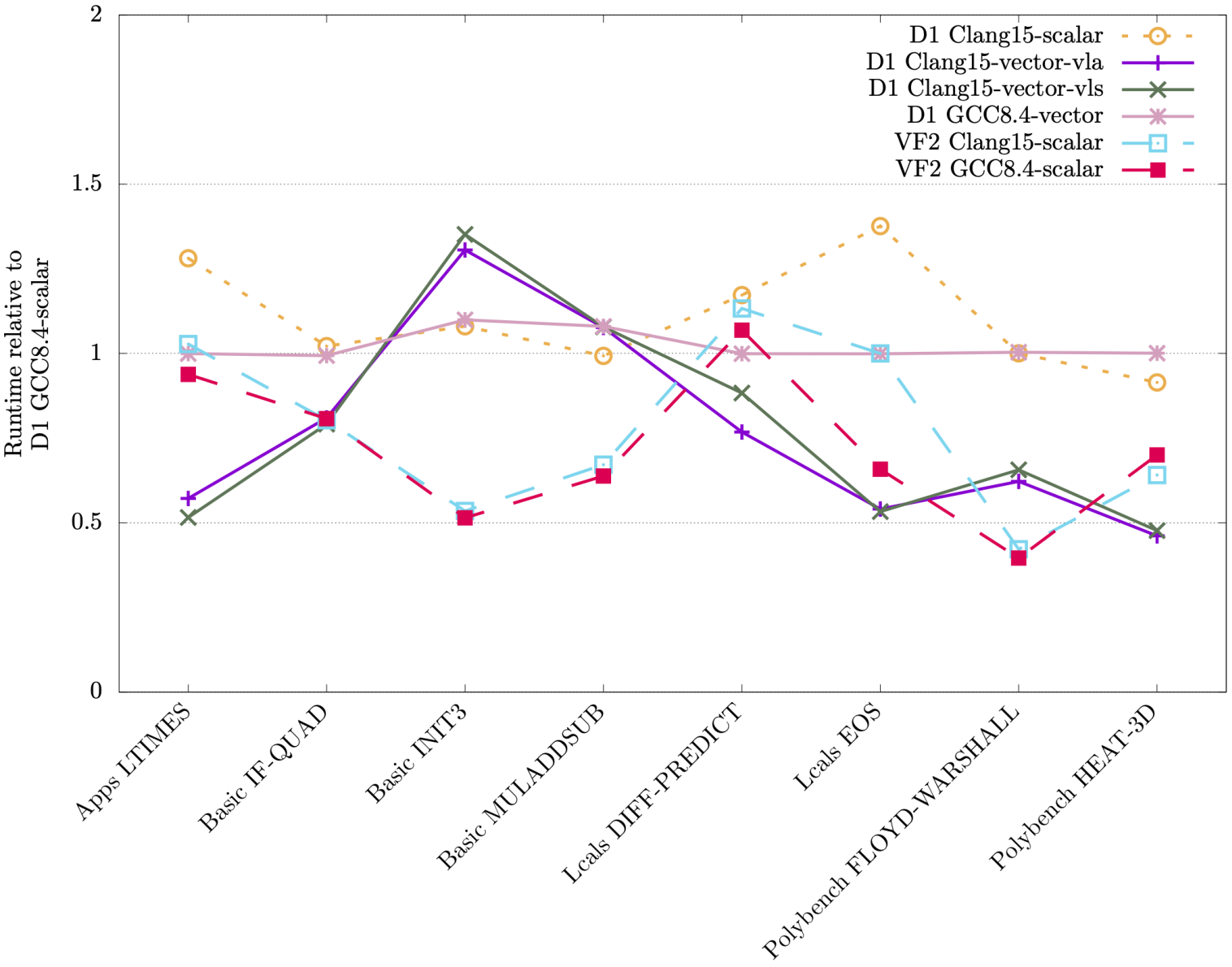
- GCC8.4-vector is VLS (for 128 bits)

Name	Compiler	RVV Version	Compiler flags
GCC8.4-scalar	XuanTie GCC8.4	N/A	-O3 -march=rv64gc -ffast-math
GCC8.4-vector	XuanTie GCC8.4	0.7	-O3 -march=rv64gcv0p7 -ffast-math
Clang15-scalar	Clang 15.0	N/A	--march=rv64gc -O3 -ffast-math
Clang15-vector-vls	Clang 15.0	1.0	-march=rv64gcv -O3 -mllvm --riscv-v-vector-bits-min=128 -ffast-math
Clang15-vector-vla	Clang 15.0	1.0	-march=rv64gcv -O3 -mllvm -scalable-vectorization=on -ffast-math

Vector Benchmark: Results

- Out of 64 kernels:
  - 30 auto-vectorised by GCC 8.4 & Clang 15-VLA & Clang 15-VLS
  - 21 auto-vectorised by Clang 15 VLA & Clang 15 VLS
  - 8 auto-vectorised by Clang 15 VLS
  - 5 not auto-vectorised at all
- -> Clang 15 vectorises more, especially VLS
- As noted in previous talk, some kernels are vectorised, but only scalar branch was executed (for both GCC and Clang)
- We chose a set of 22 kernels and translated the Clang assembly from RVV 1.0 to 0.7

Vector Benchmark: Results 1

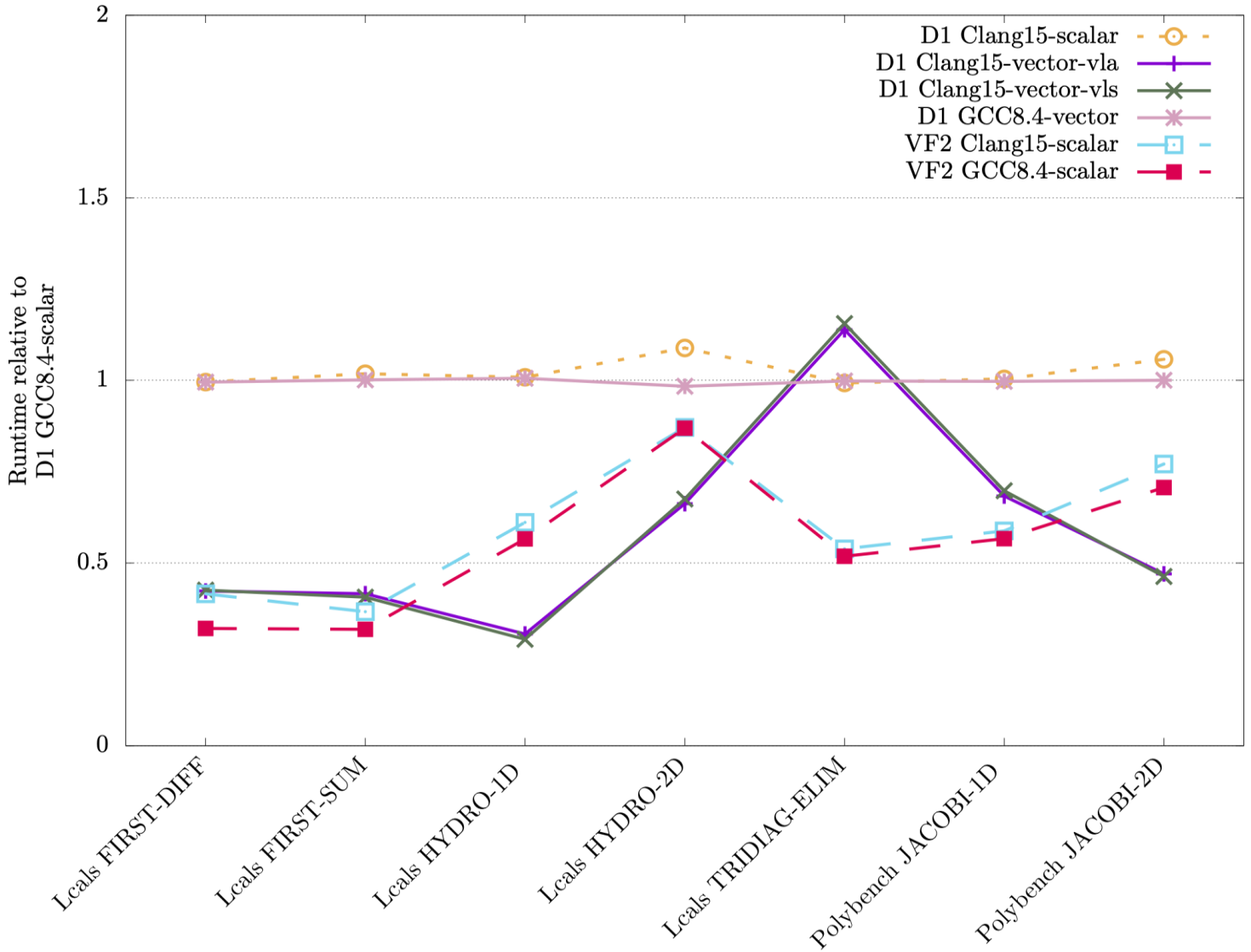


- Kernels not vectorised by GCC 8.4
  - Pink: D1 GCC-vector
  - Purple: D1 Clang-vector-vla
  - Green: D1 Clang-vector-vls
- Clang is capable of vectorising more kernels, and provide significant speedup for some
- Some kernels are slower with Clang-vector

Runtime relative to GCC 8.4-scalar on D1
Lower is better

epcc

Vector Benchmark: Results 2

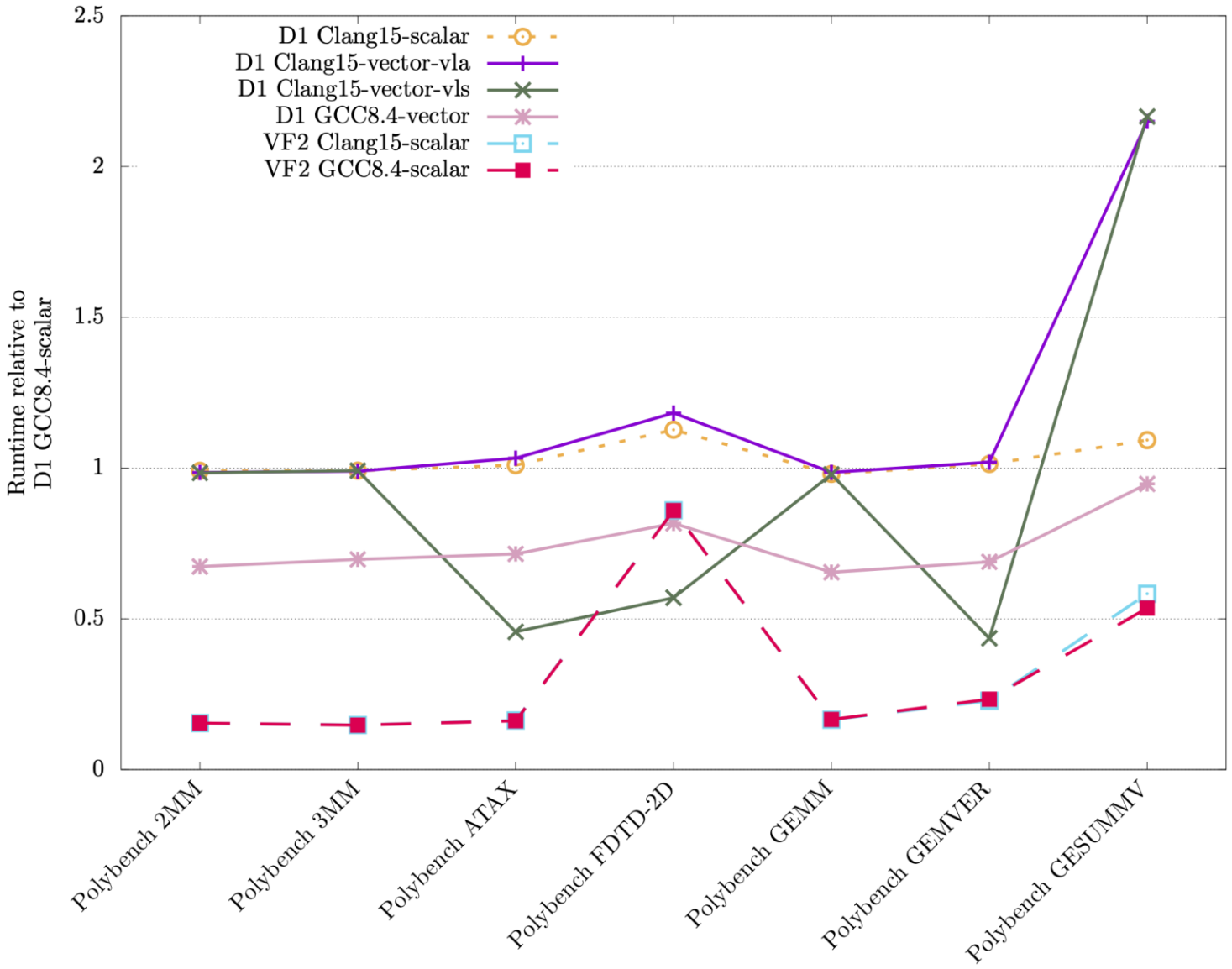


- Kernels vectorized by GCC 8.4 but only scalar code executed
  - Pink: D1 GCC-vector
  - Purple: D1 Clang-vector-vla
  - Green: D1 Clang-vector-vls
- GCC-vector ~ GCC-scalar
- Clang with VLS and VLA provide speedup

Runtime relative to GCC 8.4-scalar on D1
Lower is better

epcc

Vector Benchmark: Results 3



- Kernels vectorised by GCC and exectued
  - Pink: D1 GCC-vector
  - Purple: D1 Clang-vector-vla
  - Green: D1 Clang-vector-vls
- Matrix multiplication (GEMM…): Clang vectorised but only scalar code executed
- In some cases VLS significantly faster than VLA (e.g. ATAX)
- In some cases Clang-vector code even slower than Clang-scalar (e.g. GESUMMV)

Runtime relative to GCC 8.4-scalar on D1
Lower is better

epcc

Summary

- Explored compiler toolchains that enable vectorisation on RISC-V physical hardware
- No main branch GCC support RVV, bespoke version by T-Head with GCC8.4 does support RVV 0.7
- Less capable of vectorisation than Clang 15, which only supports RVV 1.0
- Developed *rvv-rollback* tool to backport RVV 1.0 assembly to 0.7
- Significant vectorisation performance difference between GCC, Clang VLA and Clang VLS
- Important to experiment with compiler flags and check vectorised code actually executed

Thank you!

- EPCC RISC-V Testbed: http://riscv.epcc.ed.ac.uk/

- https://github.com/RISCVtestbed/rvv-rollback