TEST-DRIVING RISC-V VECTOR HARDWARE FOR HPC

Joseph K. L. Lee, Maurice Jamieson, Nick Brown, Ricardo Jesus
Outline

• EPCC RISC-V Testbed

• RISC-V Vector Extension (RVV)

• Hardware Implementation

• Software Support: Compiler Toolchain, Linux, and Libraries

• Vector Benchmarks

• Summary & Recommendations
EPCC RISC-V Testbed

- Aim: Provide HPC code developers and data-scientists access to the latest RISC-V CPUs

- We have many boards (64 cores):

<table>
<thead>
<tr>
<th>Board</th>
<th>Processor (SOC)</th>
<th># Core</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>NezhaSTU</td>
<td>C906 (D1)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MangoPi MQ-Pro</td>
<td>C906 (D1)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>HiFive Unmatched</td>
<td>U74 (FU740)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>StarFive VisionFive V1</td>
<td>U74 (JH7100)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>StarFive VisionFive V2</td>
<td>U74 (JH7110)</td>
<td>4</td>
<td>13</td>
</tr>
</tbody>
</table>

- Also will have soft-cores
- Also have posts about building experiences
- Apply for access: [http://riscv.epcc.ed.ac.uk/](http://riscv.epcc.ed.ac.uk/)

- Funded by ExCALIBUR H&ES
RISC-V Vector Extension (RVV)

- Key feature of RISC-V: modular extensions
- Vector instructions useful in HPC applications:
  - exploit data parallelism, increase instruction bandwidth, improve energy efficiency
- Vector Extension (RVV) first proposed in 2015
- Important releases:
  - Version 0.7 (2019): stable enough to begin developing toolchains, simulators, implementations
    - (adopted by hardware e.g. C906, Vitruvius etc.)
  - Version 1.0: ratified in late 2021
RISC-V Vector Extension (RVV)

- Key features:
  - Vector length agnostic (VLA – like Arm SVE)
    - c.f. Vector Length Specific (VLS – like AVX, Arm NEON)
  - Vector length (VLEN) minimum 128 bits, up to 65,536 bits (c.f. Arm SVE max 2,048 bits)
  - Vector register grouping (LMUL): 1/2/4/8 registers
  - Fractional LMUL (not in RVV 0.7)

- Also a SIMD ‘P’ extension, aimed at embedded cores, low power DSP
  - Not yet ratified
## Vector Hardware Implementation

<table>
<thead>
<tr>
<th>Processor</th>
<th>Vector Length (bits)</th>
<th>RVV version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SiFive P270/P470/P670</td>
<td>256 / 128 / dual-128</td>
<td>1.0</td>
</tr>
<tr>
<td>SiFive X280</td>
<td>512</td>
<td>1.0</td>
</tr>
<tr>
<td>Andes NX27V</td>
<td>Configurable from 128-512</td>
<td>1.0</td>
</tr>
<tr>
<td>Andes AX45MPV</td>
<td>Configurable from 128-1024</td>
<td>1.0</td>
</tr>
<tr>
<td>Vitruvius+</td>
<td>16384</td>
<td>0.7.1 (update to 1.0 in future)</td>
</tr>
<tr>
<td>Hwacha (V4)</td>
<td>512</td>
<td>Custom</td>
</tr>
<tr>
<td>New Ara</td>
<td>Configurable, e.g. 4096</td>
<td>1.0</td>
</tr>
<tr>
<td>Tenstorrent BOOM-ocelot</td>
<td>Configurable from 128</td>
<td>1.0</td>
</tr>
<tr>
<td>T-Head XuanTie C906/C920</td>
<td>128</td>
<td>0.7.1</td>
</tr>
<tr>
<td>T-Head XuanTie C908*</td>
<td>Configurable 128/256</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Vector Hardware

- Wide range of applications:
  - General applications (SiFive P series)
  - Decoupled vector accelerator
    - Ara
    - Vitruvius+ : long vectors – 256 DP elements per register
- Off-the-shelf RVV 0.7:
  - T-Head (Alibaba) XuanTie C906
  - Found in Allwinner D1 SoC
  - 128-bit VLEN, support 8, 16, 32 bit vector elements
  - Does not support 64 bit elements(*), not suitable for HPC applications
- RVV 1.0?
  - XuanTie C908 with Sipeed?
- Softcores:
  - Some open source softcores: e.g. OpenC906, Tenstorrent Boom-ocelot
  - Requires knowledge for FPGA designs and tools
Vector Software Support: Compiler Toolchain

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*
• Results shown in upcoming talk: Backporting RISC-V vector assembly
Vector Software Support: Linux and Perf

Linux Kernel

- RISC-V Linux distribution generally available: Debian, Ubuntu, Fedora …
- Sipeed Linux image for Allwinner D1 supports RVV out of the box
  - However, bootloader is proprietary and protected, to modify Linux images must cross compile on another host, and vendor-specific patches must be applied to buildroot
  - Specific T-Head GCC compiler version must be used to ensure resulting image is RVV compatible
  - Time consuming & requires specific knowledge: high barrier to entry!

Performance Analysis & Instrumentation: Perf?

- To obtain events, kernel and OpenSBI need to be patched, depend on board & vendor
- HiFive Unmatched: the Linux kernel version 5.18 supports instruction and cycle count hardware events for perf
- Allwinner D1: official support for perf only released in Linux kernel version 6.2 on 19 Feb 2023, almost two years after the hardware was made available
- Major drawback for HPC workloads, where performance monitoring is necessary
Vector Software Support: Emulation and Libraries

Emulation

- Limited physical hardware, none yet for RVV 1.0
- QEMU, Spike: supports RVV 1.0 (earlier versions support RVV 0.7.1)
- Vehave (BSC):
  - Functional emulator based on QEMU
  - Dynamically handle and emulate vector instructions
  - Separate versions supporting RVV 1.0 and 0.7.1

Libraries

- Most HPC libraries can be cross-compiled for RISC-V, but tend to have limited vectorisation optimisation
- OpenBLAS optimised for RVV 0.7.1, requires specific compiler from T-Head (v2.6.0 toolchain)
- Effort within community to optimise libraries (e.g. FFTW)
- Likely see significantly increased support within the next year
## Vector Benchmarks: Systems

<table>
<thead>
<tr>
<th></th>
<th>Allwinner D1</th>
<th>StarFive JH7110 (VF2)</th>
<th>A64FX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>XuanTie C906</td>
<td>SiFive U74</td>
<td>Fujitsu A64FX</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.0 GHz</td>
<td>1.5 GHz</td>
<td>1.8 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>1</td>
<td>4</td>
<td>48</td>
</tr>
<tr>
<td>Cache</td>
<td>32 KB I-cache + 32 KB D-cache</td>
<td>32 KB I-cache + 32 KB D-cache + 2MB L2</td>
<td>64 KB I-cache + 64KB D-cache, 8 MB shared L2 per 12 cores</td>
</tr>
<tr>
<td>Memory</td>
<td>512MB DDR3</td>
<td>8GB DDR4</td>
<td>32GB HBM2</td>
</tr>
<tr>
<td>ISA</td>
<td>RV64GC+V0.7</td>
<td>RV64GC</td>
<td>ARMv8.2 with SVE</td>
</tr>
<tr>
<td>Vector width</td>
<td>128 bit</td>
<td>N/A</td>
<td>Dual 128-bit (NEON)/Dual 512-bit (SVE)</td>
</tr>
</tbody>
</table>
Vector Benchmark

- Only on single core
- Single precision
- For A64FX, use NEON only
  - 128-bit vector length, same as D1
  - T-Head compiler generates VLS code (fixed 128-bit)
- A64FX is designed for HPC vs RISC-V cores for embedded / single-board computer:
  - Still interesting to compare
Vector Benchmark

- Benchmark: RAJA Performance Suite (https://github.com/LLNL/RAJAPerf)
  - ALGORITHM
  - APPS
  - BASIC
  - LCALS (Livermore Compiler Analysis Loop Suite)
  - POLYBENCH
  - STREAM

<table>
<thead>
<tr>
<th>Name</th>
<th>Compiler</th>
<th>Vector width</th>
<th>Compiler flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV-GCC8.4-scalar</td>
<td>XuanTie GCC8.4</td>
<td>N/A</td>
<td>-O3 -march=rv64gc -ffastmath</td>
</tr>
<tr>
<td>RV-GCC8.4-vector</td>
<td>XuanTie GCC8.4</td>
<td>128-bit</td>
<td>-O3 -march=rv64gcv -ffastmath</td>
</tr>
<tr>
<td>ARM-GCC11.2-scalar</td>
<td>GCC 11.2</td>
<td>N/A</td>
<td>-O3 -ffastmath -mcpu=a64fx -march=armv8.2-a+nosimd+nosve</td>
</tr>
<tr>
<td>ARM-GCC11.2-vector</td>
<td>GCC 11.2</td>
<td>128-bit</td>
<td>-O3 -ffastmath -mcpu=a64fx -march=armv8.2-a+simd+nosve</td>
</tr>
</tbody>
</table>
Vector Benchmark: Results

- For RV-GCC8.4-vector, out of 64 kernels:
  - 23 vectorised and vector loop executed
  - 7 vectorised but vector loop not executed
  - 34 only scalar
  - Clang vectorises more kernel than GCC (See next talk)
  - Vectorised kernel sensitive to loop ranges, scalar branch taken often
Vector Benchmark: Results

• Summary:
  • Purple: D1-vector / D1-scalar
  • RVV achieves higher bandwidth for stream kernels
  • RVV accelerates Linear Algebra kernels:
    • 84% faster for AXPY
    • 53% faster for GEMM…
  • Speedup generally not as significant as NEON on A64FX

RISC-V timing normalised against D1 scalar
A64FX vector timing normalised against A64FX scalar
Lower is better
Vector Benchmark: Results

- Allwinner D1 vs StarFive JH7110 (VF2) (Green):
  - VF2 higher frequency, GEMM 6x faster than D1 scalar, 4x D1 w/ vector
  - But with vectorisation D1 streaming is faster than VF2, even though VF2 has higher theoretical bandwidth
  - AXPY on D1 w/ vector 77% faster than VF2 scalar
  - D1 considerably cheaper than VF2, impressive
  - But only testing 1 out of 4 cores in VF2
Summary

• D1 gains significant performance advantage with RISC-V Vector extension

• Mismatch between RVV version in available tooling (e.g. GCC and Clang) and hardware makes running and testing RVV codes difficult

• Challenges due to immaturity will hopefully be solved with standardisation of tooling and RVV 1.0 compliant hardware

• RVV provides a strong foundation for leveraging RISC-V for high performance workloads

• Improvement potentials to further increase performance:
  • improved auto-vectorisation in LLVM
  • increased VLEN in future CPUs

• Would be helpful if support present for both RVV 0.7 and 1.0 in mainstream GCC and Clang
Recommendations

• We recommend using the T-Head GCC 8.4 auto-vectorisation and *not* using the T-Head RVV v0.7 intrinsic API

• This ensures that codes can simply be recompiled, without modification, to target RVV v1.0 compatible hardware

• We also recommend building RVV-enabled Linux images with a patched mainstream buildroot using the T-Head GCC 8.4 compiler, as support for the Allwinner D1 has recently been added
Thank you!

- Next part: Backporting RISC-V Vector assembly

- EPCC RISC-V Testbed: [http://riscv.epcc.ed.ac.uk/](http://riscv.epcc.ed.ac.uk/)
Additional slides: 1

- RAJAPerf kernels vectorised by RV-GCC8.4-vector

<table>
<thead>
<tr>
<th>Vectorised and executed: Total 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Apps</td>
</tr>
<tr>
<td>Basic</td>
</tr>
<tr>
<td>Lcals</td>
</tr>
<tr>
<td>Polybench</td>
</tr>
<tr>
<td>Stream</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vectorised: Total 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lcals</td>
</tr>
<tr>
<td>Polybench</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scalar: Total 34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Apps</td>
</tr>
<tr>
<td>Basic</td>
</tr>
<tr>
<td>Lcals</td>
</tr>
<tr>
<td>Polybench</td>
</tr>
</tbody>
</table>