MXM-RVV: Easy Multicore $\times$ Multithreading with Vectors via Composable Extensions + DMA

Joseph Maheshe[1], Guy G.F. Lemieux[1]

[1]Dept of ECE, University of British Columbia

Abstract

We revisit the topic of multithreaded vector processors, but now within the RISC-V architecture. By combining data-level and thread-level parallelism, we further improve performance. Going from 1×1 MXM-RVV configuration of one vector unit and one state context to 2×2, we gain a $2.6\times$ performance boost. To do this, we first declare all RVV opcodes to be part of the Custom opode space, allowing them to be multiplexed by the Draft CX Specification [1]. Second, we add new rules to the specification to allow vector loads and stores, paying attention to coherence and consistency. Third, we create an SoC with one hart and multiple RVV vector units (multicore CXUs), where each CXU contains multiple contexts (multithreading). Fourth, we add an independent CX instruction queue to each context within each CXU, thereby enabling asynchronous multicore, multithreaded execution of vector instructions with a single scalar thread and hardware scheduling of the parallel queues.

To make this work, we show that non-blocking vector loads and stores are essential to hide memory latency by executing instructions from other contexts. We use a round-robin scheduling scheme for fine-grained multithreading, which enhances pipeline utilization for non-chained vector processors. We illustrate how to write software to target MXM-RVV and show that it requires minimal changes.

The technology has been implemented on an FPGA, using CVA5 as the out-of-order scalar processor and Saturn-V as the vector unit.

Introduction

Limits of instruction-level parallelism drive us to find more parallelism. Data-level parallelism in vectors operate on multiple elements simultaneously with a single instruction. Thread-level parallelism executes independent contexts or threads concurrently across multiple cores. This proposal combines both of these approaches to enhance the performance of one scalar RISC-V processor with multiple RISC-V vector (RVV) cores using simplified multithreading.

In this work, RVV is implemented as a Composable Extension (CX). The Draft CX Specification [1] defines a modular way for IP vendors to develop instruction-based accelerators and thereby share the otherwise limited custom opcode space available within the RISC-V ISA. CX also provides multiple contexts. This paper shows how this can be used to hide memory and pipeline stalls with thread-level parallelism.

In [2], Espasa and Valero examine the interaction between multithreading and main memory latency. They simulate a selection of Perfect Club and Specfp92 benchmarks and compare their execution time on a conventional vector architecture with a single memory port and a multithreaded vector architecture. They show that multithreading offers a performance gain of over 1.4 times for realistic memory latencies, optimizing single memory port utilization to 95 %. Modern GPUs are also multithreaded vector architectures as they interleave warps in a fine-grained manner [3].

Architecture

In MXM-RVV, after a vector instruction is decoded and confirmed as non-speculative, the scalar core forwards it to the CX interface along with any required scalar operands. Unless its result is to be written back to the scalar register file, the vector instruction is retired. Among other signals, the CX interface sends an 8b `cxu_id` and an 8b `state_ID`, both read from the custom machine-level CSR, `mcx_selector`. The `cxu_id` multiplexes up to 256 CXUs, which can be replicated vector units. Replicating an entire vector unit enables thread-level parallelism among vector instructions, similar to a chip multiprocessor offering parallelism among scalar instructions. The `state_id` provides up to 256 independent contexts, essentially replicating all state and CXU-specific CSRs. In MXM-RVV, the lower bits of `state_id` are also used to target a different instruction queue within a CXU; these queues are assumed to contain independent instructions. The multiple cores and contexts are intended to support a multiprogrammed environment. In this paper, we make use of them to enhance performance of a single-threaded program.

Non-blocking vector loads and stores

Non-blocking vector loads and stores are essential to hide memory latency with thread-level parallelism. MXM-RVV retires vector stores as soon as its data is written to a store buffer. Vector loads are split into two micro-instructions: `load-issue` and `load-commit`. Load-issue initiates memory read requests, while load-commit transfers data from the load buffer to the vector register file. Load-issue also sets a 'use' bit in a scoreboard bitmap dedicated to vector load instructions. The use bit gets cleared upon the retirement of the corresponding load-commit.

To track load-commits, a separate instruction queue is used. A vector load in the main instruction queue becomes a load-issue; when it gets to the head of the queue, it is executed and becomes a load-commit in the new queue. The load-commit is issued once the data response data arrives, thereby retiring the original vector load. Unless there is a data dependency, removing the load-issue micro-opeations from the main queue allows subsequent instructions to bypass the load-commit. The vector register file has three read ports for concurrent reads of vector ALU operands and vector store data and two write ports to ensure concurrent writing of vector ALU results and vector load data, enhancing performance.

The current Draft CX Specification lacks direct memory access and is limited to only compute-oriented custom instructions. Our work extends it to support consistent memory loads and stores. CXU loads and stores share a common DMA engine which provides unit-stride data transfers and provides fences; load-store units have been removed from their respective vector processors. Details of the shared DMA are beyond the scope of this paper.

Scheduling

Scheduling follows a round-robin scheme to select instructions ready to execute from the queues. After arbitration, we potentially have one instruction from the main queues and one from the load-commit queues. A priority arbiter then chooses the load-commit instruction over the other to retire loads and resolve data dependencies as soon as possible. The round-robin scheduling also helps interleave independent vector instructions, which improves pipeline utilization and reduces data dependency stalls.

In contrast, in chained vector processors, Espasa and Valero follow a priority scheme that favours chaining between vector instructions as much as possible [2].

Programming

To fully exploit MXM-RVV, we must partition tasks into independent subtasks to run concurrently. Listing 1 shows the vectorized `rgba2luma` benchmark, which converts a 32b RGBA image to an 8b LUMA component. There is no dependency between rows, so the vectorized benchmark interleaves the rows across up to 2 vector units and 4 state contexts. To do this, up to 4 bits are taken from the row index, `i`, using the lowest bit as `cxu_id` to multiplex the 2 vector units and the next 3 bits indicate the `state_id`. Note `vsetvli` instructions do not update the register file with `x0` as the destination register. This is intended and allows MXM-RVV to proceed and retire vector instructions without waiting for the scalar write-back, preventing potential bottlenecks.

Compared to a multiple buffering implementation, which increases code density linearly, minimal changes to the original code are required as we add only three statements inside the loop with `partition_row()`. Although we use inline assembly and assume we have machine-level privileges to update the CSR, ongoing work on the CX API and ABI aims to simplify programming with machine, user, and supervisor-level privileges.

Evaluation

MXM-RVV uses the CVA5 [4] scalar processor and the Saturn-V [5] vector unit. Optimized for FPGAs, the CVA5 executes RV32IM instructions out-of-order. Saturn-V implements the RVV-Lite ISA, an extensible subset of RISC-V vectors, with a 64b width for execution and memory. To reduce memory latency, we have added AXI4 support with burst transfers.

In the test system, CVA5 has a 16 kB instruction cache and 64 kB data cache. The vectorized version targets MXM-RVV with a VLEN of 1024. Saturn-V constraints the ratio SEW/LMUL to 8 [6], which results in a VLMAX of $1024/8 = 128$. We run the `rgba2luma` benchmark 100 times with a 128×128 image size. As for MXM-RVV's instruction queue depths, main queues have a depth of 64, while load-commit queues have a depth of 1.

Table 1 reports the results using various configurations of MXM-RVV shown as m vector units $\times$ n state contexts. We compare the average cycle count and vector ALU utilization across 100 independent runs. We define Vector ALU utilization as the ratio of the ALU's busy cycles to its total cycles (busy + idle). A low utilization indicates that ALU instructions are not being processed to hide memory latency, whereas a utilization value 1.0 means memory latency is completely hidden.

The baseline scalar code achieves an IPC of 0.80 with a cold cache. After 100 iterations to warm up the cache, IPC increases to 0.99, which gives a 1.2× speedup. Run with a cold cache, the vectorized version shows improvements in cycle count with increasing vector units and state contexts; the baseline 1×1 vector configuration is **`5.0x`** faster than scalar code.

```c
// EXTRACT n bits starting at position i from x
#define GET_BITS(x, n, i) (((x) >> (i)) & ((1 << (n)) - 1))

// UNIQUE SoC PARAMETERS
#define LOG2_NUM_CXUS 1
#define NUM_CXUS (1<<LOG2_NUM_CXUs)
#define LOG2_NUM_STATES 2
#define NUM_STATES (1<<LOG2_NUM_STATES)

// GENERAL CXU CONTROL
#define CSR_MCX_SELECTOR 0xBC0
#define CSR_CX_STATUS    0x801
#define MCX_SHAMT_CXU_ID    0
#define MCX_SHAMT_STATE_ID  16
#define MCX_SHAMT_CXE       28
#define MCX_SHAMT_VERSION   29
/* m-mode write to selector CSR using cxu_id and state_id */
#define MCX_SELECT(cxu_id, state_id) \
    asm volatile ("csrw %[csr], %[rs];" :: \
    [rs]  "r" ((1 << MCX_SHAMT_VERSION) |  /* enable muxing */ \
              (0  << MCX_SHAMT_CXE) | /* disable exceptions */ \
              (state_id << MCX_SHAMT_STATE_ID) | \
              (cxu_id   << MCX_SHAMT_CXU_ID)), \
    [csr] "i" (CSR_MCX_SELECTOR));
#define CSRR_CX_STATUS(status) \
    asm volatile ( "csrrc %[csr], %[rd], x0;" :: \
    [rd]  "w" (status), \
    [csr] "i" (CSR_CX_STATUS) );

void partition_row(uint32_t row) {
    // stripes row across (NUM_STATES * NUM_CXUS) async threads
    int cxu_id   = GET_BITS(row, LOG2_NUM_CXUS, 0);
    int state_id = GET_BITS(row, LOG2_NUM_STATES, 0+LOG2_NUM_CXUS);
    MCX_SELECT( cxu_id, state_id );
}

// MICROBENCHMARK
void rgba2luma(uint8_t *luma, uint32_t *rgb,
              const uint32_t IMG_W, const uint32_t IMG_H) {

  for ( uint32_t i = 0, vl; i < IMG_H; ++i ) {
    //////////////////////////////////////////////////////////////
    // this block contains the only changes to the microbenchmark
    partition_row(i);
    //////////////////////////////////////////////////////////////

    assert( IMG_W <= 4*RV_VLEN/32 ); // simplifying assumption
    asm volatile ("vsetvli x0, %[REQ_VL], e16, m2, ta, mu"
            :: [REQ_VL]  "r" (IMG_W));
    // computes {R8[j],G8[j],B8[j]} = RGB32[j]
    asm volatile ("vle32.v v4, (%0)":: "r"(rgb));
    asm volatile ("vnsrl.wi  v8,  v4, 16");
    asm volatile ("vnsrl.wi v10,  v4, 8");
    asm volatile ("vnsrl.wi v12,  v4, 0");
    asm volatile ("vand.vx   v8,  v8, %0":: "r"(255));
    asm volatile ("vand.vx  v10, v10, %0":: "r"(255));
    asm volatile ("vand.vx  v12, v12, %0":: "r"(255));
    // computes LUMA16[j] = 66*R8[j] + 129*G8[j] + 25*B8[j]+128
    asm volatile ("vmul.vx   v8,  v8, %0":: "r"(66));
    asm volatile ("vmul.vx  v10, v10, %0":: "r"(129));
    asm volatile ("vmul.vx  v12, v12, %0":: "r"(25));
    asm volatile ("vadd.vv  v14,  v8, v10");
    asm volatile ("vadd.vx  v16, v12, %0":: "r"(128));
    asm volatile ("vadd.vv  v14, v14, v16");
    // computes LUMA8[j] = LUMA16[j]>>8
    asm volatile ("vsetvli x0, x0, e8, m1, ta, mu");
    asm volatile ("vnsrl.wi v14, v14, 8");
    asm volatile ("vse8.v   v14, (%0)" : "+r" (luma));

    luma += IMG_W;
    rgb  += IMG_W;
  }
  // stalls here to sync ALL async threads among all CXUS
  uint32_t status;
  CSRR_CX_STATUS(status); // reads cx_status CSR, ignores result
}
```

Listing 1: *Example benchmark leveraging MXM-RVV architecture with 2 vector units and 4 state contexts each*

Config.	Cycles (k)	VALU Util. (%)
scalar (c)	469.1 (0.20×)	0
scalar (w)	380.1 (0.24×)	0
1 × 1	92.5 (1.0×)	71.4
1 × 2	69.3 (1.3×)	95.6
1 × 4	68.7 (1.3×)	96.4
1 × 8	69.6 (1.3×)	95.1
2 × 1	46.6 (2.0×)	71.4, 71.4
2 × 2	35.1 (2.6×)	95.3, 95.5
2 × 4	34.8 (2.7×)	95.9, 96.9
2 × 8	35.2 (2.6×)	94.6, 96.7

Table 1: *Performance of* `rgba2luma` *on MXM-RVV with m vector units × n state contexts, each with VLEN=1024. (c) = cold cache; (w) = warm cache.*

Beyond this, MXM-RVV shows up to 2.7× additional speedup through the use of additional vector units and contexts. The configurations with two vector units are nearly twice the speed of those with one unit. Note this extra speedup is achieved with only the 3 lines marked inside the loop (and checking CX_STATUS at the end). We note performance and ALU utilization plateau as the number of contexts reaches 4 and start to degrade after. With two vector units, there is no additional performance after 4 contexts because of structural conflicts to commit loads as the memory controller becomes saturated. Overall, the 2×2 vector version is 13.4× faster than the cold cache scalar code.

Status and Limitations

MXM-RVV is implemented on the ZCU104 evaluation kit with $512M \times 8b$ DDR4 SDRAM and running at 200 MHz. It has unit-stride but no strided or indexed memory operations. Within a state context, only one memory load or store can be outstanding, although multiple memory operations from different contexts can be concurrent. Additionally, MXM-RVV operates as a non-coherent accelerator, bypassing the scalar cache. This typically requires flushing the accelerator's data region from the cache before execution [7]. CVA5's write-through cache policy eliminates that requirement. Finally, MXM-RVV only supports M-mode without virtual memory.

References

[1] J. Gray, G. Lemieux, et al. *Draft Proposed RISC-V Composable Custom Extensions Specification.* Version 0.95.240403.

[2] R. Espasa and M. Valero. “Multithreaded vector architectures”. In: *HPCA*. Feb. 1997, pp. 237–248.

[3] O. Mutlu. *Fined-Grained Multithreading Video*. Jan. 2021.

[4] Eric Matthews and Lesley Shannon. “TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features”. In: *FPL*. Sept. 2017, pp. 1–4.

[5] G. Lemieux. “Poster: RVV-lite v0.5: A Modest Proposal for Reducing the RISC-V Vector Extension”. In: *RISC-V Summit Europe*. June 2023. URL: `https://riscv-europe.org/summit/2023/posters`.

[6] C. White. “Design and implementation of RVV-Lite: a layered approach to the official RISC-V vector ISA”. MA thesis. University of British Columbia, 2023.

[7] D. Giri, P. Mantovani, and L. Carloni. “Accelerators and Coherence: An SoC Perspective”. In: *IEEE Micro* 38.6 (Nov. 2018), pp. 36–45.