1 Introduction and goals

The goal of this laboratory assignment is to allow you to explore the RISC-V vector ISA using its functional simulator, Spike.

Students will write RISC-V vector assembly code to gain a better understanding of how data-level parallel code maps to vector-style processors, and to practice optimizing vector code for a given implementation. For the open-ended section, students will optimize one of three vector implementations.

The lab has two sections, a directed portion and an open-ended portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The open-ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task.

For both the directed portion and the open-ended portion, students can work individually or in groups of two (not three). Students are encouraged to discuss solutions to the lab assignments with other groups, but each group must run through the lab by themselves and turn in their own lab report.

For this lab, there are three open-ended questions, and each group is supposed to do only one of them. If you would prefer to do something else, you must contact your TA or professor with an alternate proposal of significant rigor.

2 Background

2.1 Example: Conditionalized Single-precision A · X Plus B (CSAXPY)

The RISC-V vector ISA programming model is best explained by contrast with other, popular data-parallel assembly programming models. As a running example, we use a conditionalized SAXPY kernel, CSAXPY. Figure 1 shows CSAXPY expressed in C as both a vectorizable loop and as a SPMD kernel. CSAXPY takes as input an array of conditions, a scalar $a$, and vectors $x$ and $y$, and then it computes $y += ax$ for the elements for which the condition is true.
Figure 1: **Conditional SAXPY kernel written in C.** The SPMD kernel launch code for (b) is omitted for brevity.

2.2 Packed SIMD Assembly Programming Model

Figure 2 shows CSAXPY kernel mapped to a hypothetical packed SIMD architecture, similar to Intel’s SSE and AVX extensions. This SIMD architecture has 128-bit registers, each partitioned into four 32-bit fields. As with other packed SIMD machines, ours cannot mix scalar and vector operands, so the code begins by filling a SIMD register with copies of \( a \) (Line 2). To map a long vector computation to this architecture, the compiler generates a *stripmine loop*, each iteration of which processes one four-element vector. In this example, the stripmine loop consists of a load from the conditions vector (Line 6), which in turn is used to set a predicate register (Line 7). The next four instructions (Line 8 - 11), which correspond to the body of the *if*-statement in Figure 1a, are masked by the predicate register\(^1\). Finally, the address registers are incremented by the SIMD width (Line 13 - 14), and the stripmine loop is repeated until the computation is finished (Line 15) —almost. Since the loop handles four elements at a time, extra code is needed to handle up to three *fringe* elements. For brevity, we omitted this code; in this case, it suffices to duplicate the loop body, predating all of the instructions on whether their index is less than \( n \).

The most important drawback to packed SIMD architectures lurks in the assembly code: the SIMD width is expressly encoded in the instruction opcodes and memory addressing code. When the architects of such an ISA wish to increase performance by widening the vectors, they must add a new set of instructions to process these vectors. This consumes substantial opcode space: for example, Intel’s newest AVX instructions are as long as 11 bytes. Worse, application code cannot automatically leverage the widened vectors. In order to take advantage of them, application code must be recompiled. Conversely, code compiled for wider SIMD registers fails to execute on older machines with narrower ones. As we later show, this complexity is merely an artifact of poor design.

\(^1\)We treat packed SIMD architectures generously by assuming the support of full predication. This feature is quite uncommon. Intel’s AVX architecture, for example, only supports predication as of 2015, and then only in its Xeon line of server processors.
csaxpy_simd:
  slli a0, a0, 2
  add a0, a0, a3
  vsplat4 vv0, a2
stripmine_loop:
  vl4 vv1, (a1)
  vcmpez4 vp0, vv1
!vp0 vl4 vv1, (a3)
!vp0 vl4 vv2, (a4)
!vp0 vfma4 vv1, vv0, vv1, vv2
!vp0 vs4 vv1, (a4)
  addi a3, a1, 4
  addi a3, a3, 16
  addi a4, a4, 16
  bleu a3, a0, stripmine_loop
# handle edge cases
# when (n % 4) != 0 ...
  ret

Figure 2: CSAXPY kernel mapped to the packed SIMD assembly programming model. In all pseudo-assembly examples presented in this section, a0 holds variable n, a1 holds pointer cond, a2 holds scalar a, a3 holds pointer x, and a4 holds pointer y.

csaxpy_simt:
  mv t0, tid
  bgeu t0, a0, skip
  add t1, a1, t0
  lb t1, (t1)
  beqz t1, skip
  slli t0, t0, 2
  add a3, a3, t0
  add a4, a4, t0
  lw t1, (a3)
  lw t2, (a4)
  fma t0, a2, t1, t2
  sw t0, (a4)
  skip:
  stop

Figure 3: CSAXPY kernel mapped to the SIMT assembly programming model.
2.3 SIMT Assembly Programming Model

Figure 3 shows the same code mapped to a hypothetical SIMT architecture, akin to an NVIDIA GPU. The SIMT architecture exposes the data-parallel execution resources as multiple threads of execution; each thread executes one element of the vector. One inefficiency of this approach is immediately evident: the first action each thread takes is to determine whether it is within bounds, so that it can conditionally perform no useful work. Another inefficiency results from the duplication of scalar computation: despite the unit-stride access pattern, each thread explicitly computes its own addresses. (The SIMD architecture, in contrast, amortized this work over the SIMD width.) Moreover, massive replication of scalar operands reduces the effective utilization of register file resources: each thread has its own copy of the three array base addresses and the scalar a. This represents a threefold increase over the fundamental architectural state.

2.4 Traditional Vector Assembly Programming Model

Packed SIMD and SIMT architectures have a disjoint set of drawbacks: the main limitation of the former is the static encoding of the vector length, whereas the primary drawback of the latter is the lack of scalar processing. One can imagine an architecture that has the scalar support of the former and the dynamism of the latter. In fact, it has existed for over 40 years, in the form of the traditional vector machine, embodied by the Cray-1. The key feature of this architecture is the vector length register (VLR), which represents the number of vector elements that will be processed by the vector instructions, up to the hardware vector length (HVL). Software manipulates the VLR by requesting a certain application vector length (AVL); the vector unit responds with the smaller of the AVL and the HVL. As with packed SIMD architectures, a stripmine loop iterates until the application vector has been completely processed. But, as Figure 4 shows, the difference lies in the manipulation of the VLR at the head of every loop iteration (Line 3). The primary benefits of this architecture follow directly from this code generation strategy. Most importantly, the scalar software is completely oblivious to the hardware vector length: the same code executes correctly and with maximal efficiency on machines with any HVL. Second, there is no fringe code: on the final trip through the loop, the VLR is simply set to the length of the fringe.

The advantages of traditional vector architectures over the SIMT approach are owed to the coupled scalar control processor. There is only one copy of the array pointers and of the scalar a. The address computation instructions execute only once per stripmine loop iteration, rather than once per element, effectively amortizing their cost by a factor of the HVL.

Figure 5 shows a diagram of the programmer’s view of a traditional vector processor. The vector processor is composed of a control processor and a vector of microthreads. The control processor fetches, decodes, and executes regular scalar code. It also fetches and decodes vector instructions, translating and sending the appropriate vector commands to an attached vector unit, which is conceptually composed on a vector of microthreads.

A typical sequence of traditional vector assembly code is shown on the right half of Figure 5.

2.5 RISC-V Vector ISA

One killer feature of the RISC-V Vector ISA compared to the traditional vector machine is polymorphic vector shapes and types. We can load scalar, vector, or matrix values of different types (e.g. integer or floating point with different widths) on vector registers. By setting configurations for each vector register, we can use the same instruction for the same operation on different shapes.
Figure 4: CSAXPY kernel mapped to the traditional vector assembly programming model.

Figure 5: The programmer's view of a traditional vector processor.
csaxpy_tvec:
    # configuration:
    # v0: scalar int width 8 bits
    # v1: vector int width 8 bits (vector masks)
    # v2: vector int width 8 bits
    # v3: scalar float with 32 bits
    # v4-v5: vector float with 32 bits
    setvcfg(vcfg0,
        SCALAR | INT | W8,
        VECTOR | INT | W8,
        VECTOR | INT | W8,
        SCALAR | FP | W32)
    setvcfg(vcfg2,
        VECTOR | FP | W32,
        VECTOR | FP | W32,
        0, 0)
stripmine_loop:
    setvl(t0, a0)
    vinsert v0, x0, x0 # v0[0] = 0
    vld v2, 0(a1) # load cond[i]
    vsne v1, v2, v0 # set if cond[i] != 0
    vinsert v3, a2, x0 # v2[0] = a
    vld v4, 0(a3), v1t # load x[i] if cond[i] != 0
    vld v5, 0(a4), v1t # load y[i] if cond[i] != 0
    vmadd v5, v3, v4, v5, v1t # y[i] = a * x[i] + y[i] if cond[i] != 0
    vst v5, 0(a4), v1t # store y[i] if cond[i] != 0
    add a1, a1, t0 # bump cond
    sll t1, t0, 2 # byte offset
    add a3, a3, t1 # bump x
    add a4, a4, t1 # bump y
    sub a0, a0, t0 # decrement n
    bnez a0, stripmine_loop # loop
    ret

Figure 6: CSAXPY kernel mapped to the RISC-V Vector ISA.
and types. For example, we can use \texttt{vadd} for the addition on two integer vectors, or the addition on a floating-point scalar and a floating-point vector.

Figure 6 shows the CSAXPY kernel implemented with the RISC-V Vector ISA. Not that there is no separate vector predicate registers. Instead, \texttt{v1} serves as a predicate register for vector masks, which can be set by \texttt{vseq}, \texttt{vsne}, \texttt{vslt}, or \texttt{vsge}. By annotating each instruction with \texttt{vit} (or \texttt{vif}) (Line 23 – 26), the instructions are executed conditionally in case the LSBs of each element in \texttt{v1} are one (or zero).

### 2.6 Graded Items

You will turn in a hard copy of your results to the professor or TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

1. (Directed) Problem 3.3: Compare vectorized CSAXPY (\texttt{vec-scaxpy}) against scalar CSAXPY (\texttt{csaxpy}).
2. (Directed) Problem 3.4: Implement vectorized Single-precision GEneralized Matrix-Vector multiply (\texttt{vec-sgemv}).
3. (Directed) Problem 3.5: Implement vectorized Double-precision GEneralized Matrix Multiply (\texttt{vec-dgemm}).
4. (Directed) Problem 3.6: Implement vectorized Complex Multiply (\texttt{vec-cmplxmult}).
5. (Directed) Problem 3.7: implement vectorized Index of MAX (\texttt{vec-imax}).
6. (Open-ended) Problem 4.1: implement and optimize vectorized Sparse Matrix Multiply (\texttt{vec-spmv}).
7. (Open-ended) Problem 4.2: implement and optimize vectorized Radix Sort (\texttt{vec-rsort}).
8. (Open-ended) Problem 4.3: implement and optimize vectorized 3x3 Convolution (\texttt{vec-conv}).

Also, you are supposed to complete the following table for each question:

<table>
<thead>
<tr>
<th></th>
<th>\texttt{csaxpy}</th>
<th>\texttt{sgemv}</th>
<th>\texttt{dgemm}</th>
<th>\texttt{cmplxmult}</th>
<th>\texttt{imax}</th>
<th>one of spmv, rsort, conv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar (CPI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar (GFLOs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector (CPI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector (GFLOPs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Performance of Floating-point Benchmarks.

Also, submit your code for all problems except Problem 3.3 to Donggyu through email (dgkim@eecs.berkeley.edu).
3 Directed Portion

3.1 General Methodology

This lab will focus on writing code for vector machines with the RISC-V Vector ISA. This will be done in two steps: 1) write RISC-V Vector assembly code for each benchmark, and 2) test its correctness and estimate its performance using the RISC-V ISA simulator, Spike.

Spike is a functional simulator, which does not compute the performance of program execution. However, in this lab, a simple timing model is introduced with the following assumptions:

- Single-issue in-order scalar processor
- Vector processor with 8 elements and 8 lanes
- Perfect branch prediction.
- ALU: latency = 1 cycle.
- FPU: latency = 4 cycles, fully-pipelined
- L1 data cache: latency = 3 cycles, size = 32KiB, fully-pipelined
- Main memory: latency = 20 cycles

Because all functional units are fully-pipelined, instructions can be issued every cycle when there is no data dependency. However, the processor needs to stall when instructions cannot be issued due to data dependencies or cache misses.

3.2 Setting Up Your Workspace

To complete this lab you will log in to an instructional server (icluster6-9.eecs.berkeley.edu). First, clone the lab repo and move to the benchmark directory:

```
inst$ cd ~
inst$ source ~cs152/sp18/cs152.barshrc # Can be added to ~/.bash_profile
inst$ git clone ~cs152/sp18/lab4.git
inst$ cd lab4/benchmarks
```

Run `make` to generate binaries and disassembly dump files for all benchmarks, and run `make run` to execute all the benchmarks in Spike, which will fail for unimplemented benchmarks for now. You can also run the following commands for each benchmark:

```
inst$ make <benchmark>.riscv # compile the binary for <benchmark>
inst$ make <benchmark>.riscv.dump # generate disassembly for <benchmark>
inst$ make <benchmark>.riscv.out # commit log trace for <benchmark>
```

Note that `<benchmark>.riscv.out` contains commit log traces, which is useful for debugging your code when it fails (See Appendix A for more information about debugging.)
3.3 Comparing vectorized CSAXPY (vec-csaxpy) against scalar CSAXPY (csaxpy)

For this question, you will compare the performance of vectorized CSAXPY (vec-csaxpy) against that of scalar CSAXPY (csaxpy). First, run csaxpy with the following command:

```plaintext
inst$ make csaxpy.riscv.out
spike --isa=rv64gcv --dc=128:4:64 -l csaxpy.riscv 2> csaxpy.riscv.out
mcycle = ...
minstret = ...
mpmcounter3 = ... # FLOP
mpmcounter4 = ... # D$ accesses
mpmcounter5 = ... # D$ misses
```

It will execute csaxpy.riscv in Spike, report the performance stats, and dump the commit log trace to csaxpy.riscv.out.

In addition, you should add code in vec-csaxpy/vec-csaxpy.S for the vectorized SAXPY. You can copy and paste the code in Figure 6, or write your own code. Then, run the following command:

```plaintext
inst$ make vec-csaxpy.riscv.out
spike --isa=rv64gcv --dc=128:4:64 -l vec-csaxpy.riscv 2> vec-csaxpy.riscv.out
mcycle = ...
minstret = ...
mpmcounter3 = ... # FLOP
mpmcounter4 = ... # D$ accesses
mpmcounter5 = ... # D$ misses
```

Report the cycle counts for both SAXPYs and the speedup of the vectorized SAXPY. If the speedup is less than MVL (=8), what do you think the reason?

3.4 Vectorizing Double precision GEneralized Matrix-Vector multiply (dgemv)
TBD

3.5 Vectorizing Double precision Generalized Matrix-Matrix multiply (dgemm)
TBD

3.6 Vectorizing Complex Multiply (complxmult)
TBD

3.7 Vectorizing Index of MAX (imax)
TBD
4 Open-ended Portion (+2 extra points)

For this lab, there are three open-ended questions, one of which only need to be done by each group (one or two students). Note that at least one group per question will get extra points, and thus, the more challenging the question is, the more likely your will get extra points. (Because there will be only few groups hunting that question!)

4.1 Contest: Vectorizing and Optimizing Sparse Matrix-Vector multiply (SpMV)
TBD.

4.2 Contest: Vectorizing and Optimizing Radix Sort (rsort)
TBD.

4.3 Contest: Vectorizing and Optimizing 3 x 3 Convolution (conv)
TBD.

5 The Third Portion: Feedback

This is the first lab with the RISC-V Vector ISA, and as such, we would like your feedback! Please fill out the survey form at http://tinyurl.com/cs152-sp18-lab4-survey.

How many hours did the directed portion take you? How many hours did you spend on the open-ended portion? Was this lab boring? Did you learn anything? Is there anything you would change? Feel free to write as little or as much as you want.

6 Acknowledgments

A Appendix: Debugging

TBD.