C152 Laboratory Exercise 4 (Version 1.1)

Professor: Krste Asanović
TA: David Biancolin
Department of Electrical Engineering & Computer Science
University of California, Berkeley

April 5, 2019

Changelog

Version 1.1:

- Remove CPI-based "Speed Up" from table.
- Added a link to the v0.4 RVV specification (Chapter 18).

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. This lab uses an older version of the vector extension (version 0.4, see Chapter 18 of this PDF [1]).

In this lab, you 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, you'll write an optimized vector implementation for one two kernels: Sparse Matrix-Vector Multiply (*SpMV*) or radix sort (*rsort*).

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 or 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.

In this lab, there are two open-ended questions but each group need only do one. If you'd like to do something else, contact your TA or professor with an alternate proposal of significant rigor.

[1] https://www-inst.eecs.berkeley.edu/ cs152/sp19/handouts/sp19/riscv-spec-rvv-v0p4.pdf

2 Background

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

The RISC-V vector ISA programming model is best explained by contrasting it 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.

```
1 void csaxpy(size_t n, bool cond[], float a, float x[], float y[])
2 {
3   for (size_t i = 0; i < n; ++i)
4     if (cond[i])
5       y[i] = a*x[i] + y[i];
6 }
```

(a) vectorizable loop

```
1 void csaxpy_spmd(size_t n, bool cond[], float a, float x[], float y[])
2 {
3   if (tid < n)
4     if (cond[tid])
5       y[tid] = a*x[tid] + y[tid];
6 }
```

(b) SPMD kernel

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[2]. 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

[2]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:
     vlb4    vv1, (a1)
     vcmpez4 vp0, vv1
!vp0 vlw4    vv1, (a3)
!vp0 vlw4    vv2, (a4)
!vp0 vfma4   vv1, vv0, vv1, vv2
!vp0 vsw4    vv1, (a4)
     addi    a1, 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.**

to three *fringe* elements. For brevity, we omitted this code; in this case, it suffices to duplicate the loop body, predicating 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.

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 that 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. One attractive alternative is a *traditional* vector architecture, 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.

```
1  csaxpy_tvec:
2  stripmine_loop:
3       vsetvl   t0, a0
4       vlb      vv0, (a1)
5       vcmpez   vp0, vv0
6  !vp0 vlw      vv0, (a3)
7  !vp0 vlw      vv1, (a4)
8  !vp0 vfma     vv0, vv0, a2, vv1
9  !vp0 vsw      vv0, (a4)
10      add      a1, a1, t0
11      slli     t1, t0, 2
12      add      a3, a3, t1
13      add      a4, a4, t1
14      sub      a0, a0, t0
15      bnez     a0, stripmine_loop
16      ret
```

Figure 4: **CSAXPY kernel mapped to the traditional vector assembly programming model.**

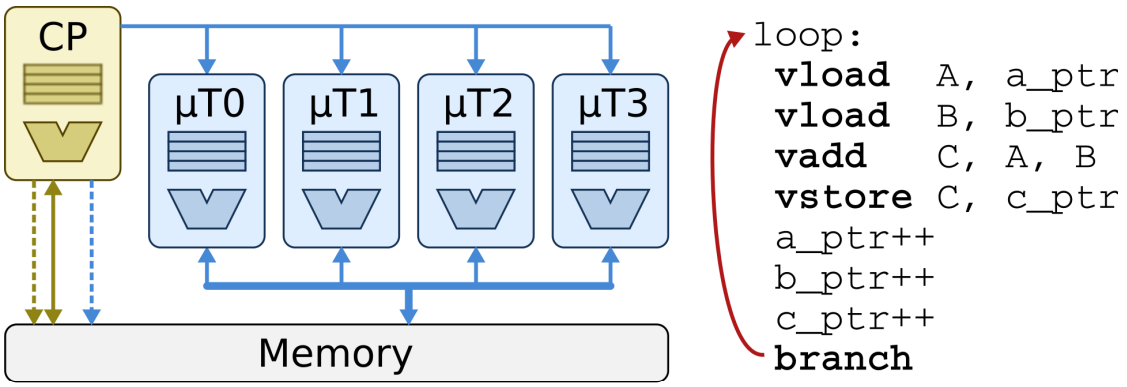

Figure 5: The programmer's view of a *traditional* vector processor.

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 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 and types. For

```
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.**

example, we can use `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, `v1` serves as a predicate register for vector masks, which can be set by `vseq`, `vsne`, `vslt`, or `vsge`. By annotating each instruction with `v1t` (or `v1f`) (Line 23 – 26), the instructions are executed conditionally in case the LSBs of each element in `v1` are one (or zero).

2.6 Graded Items

You will turn in a digital copy of your lab over gradescope. 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 (`vec-scaxpy`) against scalar CSAXPY (`csaxpy`).
2. (Directed) Problem 3.4: Implement vectorized Single-precision GEneralized Matrix-Vector multiply (`vec-sgemv`).
3. (Directed) Problem 3.5: Implement vectorized Double-precision GEneralized Matrix Multiply (`vec-dgemm`).
4. (Directed) Problem 3.6: Implement vectorized Complex Multiply (`vec-cmplxmult`).
5. (Directed) Problem 3.7: implement vectorized Index of MAX (`vec-imax`).
6. (Open-ended) Problem 4.2: implement and optimize vectorized Sparse Matrix Multiply (`vec-spmv`).
7. (Open-ended) Problem 4.3: implement and optimize vectorized Radix Sort (`vec-rsort`).

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

	`csaxpy`	`sgemv`	`dgemm`	`cmplxmult`	`imax`
Scalar (CPI)					
Scalar (FLOPs/cycle)					
Vector (CPI)					
Vector (FLOPs/cycle)					
Speedup (FLOPs/cycle)					

Table 1: Performance of Floating-point Benchmarks.

Your directed and open-ended reports must not exceed **10 pages**, each, excluding front matter. Submissions that exceed this length will be penalized 1 point.

3 Directed Portion (3 Points)

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, line size = 64 Bytes, 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/sp19/cs152.lab4.bashrc # Can be added to ~/.bash_profile
inst$ git clone ~cs152/sp19/lab4.git
inst$ cd lab4/benchmarks
```

Also, for a new release, you can pull it in with:

```
inst$ cd ${LAB4ROOT}
inst$ git pull origin master
```

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:

```
inst$ make csaxpy.riscv.out
spike --isa=rv64gcv --dc=128:4:64 -l csaxpy.riscv 2> csaxpy.riscv.out
    cycles       = ...
    instructions = ...
    FLOPs        = ...
    D$ accesses  = ...
    D$ misses    = ...
```

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

Next, run simulation for vector CSAXPY:

```
inst$ make vec-csaxpy.riscv.out
spike --isa=rv64gcv --dc=128:4:64 -l vec-csaxpy.riscv 2> vec-csaxpy.riscv.out
    cycles       = ...
    instructions = ...
    FLOPs        = ...
    D$ accesses  = ...
    D$ misses    = ...
```

Collect your results and fill out the column of `csaxpy` in Table 1. Compute each row with the following formulas:

- `CPI` = `cycles` / `instructions`
- FLOPs/cycle = `FLOPs` / `cycles`
- Speedup = performance of vectorized version / performance of scalar version

In your report, explain the performance difference between the two implementations. Make reference to the source and each implementation's commit log as necessary to support your explanation.

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

Now that you understand the infrastructure, how to run benchmarks, and how to collect results, you can write your own benchmark and measure its performance.

For Problem 3.4, you will vectorize Double Precision Generalized Matrix-Vector Multiply (`dgemv`), which is a fundamental kernel for scientific computing (Basic Linear Algebra Subprogram (BLAS) Level 2). Its (unoptimized) pseudo-code is shown below:

```
// pseudo code
for (i = 0 ; i < m ; i++)
{
  y[i] = 0.0;
  for (j = 0 ; j < n ; j++)
  {
    y[i] += A[i][j] * x[j];
  }
}
```

The scalar version is provided in `$LAB4ROOT/benchmarks/dgemv`. To measure its performance, run:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make dgemv.riscv.out
```

Your goal is to vectorize the inner loop of `dgemv` in `$LAB4ROOT/benchmarks/vec-dgemv/vec-dgemv-inner.S`. There are blanks (`TODO`s) for vector instructions as well as a brief description of the RISC-V ABI calling convention (which provides suggestions on which registers to use).

When you are ready to test your code, run it on the ISA simulator

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make vec-dgemv.riscv.out
```

If no errors are reported, you're done! Collect your results from the simulations and fill out the corresponding entries in Table 1.

Hints: you need to do a reduction for this question. For this, you may find the following instruction to be particularly handy.

- `vslide vd, vs1, rs2`
  `vd[i] := 0 ≤ (rs2) + i < VL ? vs1[(rs2) + i] : 0.`

In your report, provide only snippet of code *you* wrote, explain the performance difference between the scalar and your vector implementation.

3.5 Vectorizing Double precision GEneralized Matrix-Matrix multiply (`dgemm`)

For Problem 3.5, you will vectorize Double Precision Generalized Matrix Multiply (`dgemm`), which is another fundamental kernel for scientific computing and machine learning (BLAS Level 3). Its *unoptimized* pseudo-code is shown below:

```
// pseudo code
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
     for ( k = 0 ; k < n ; k++ )
       C[i][j] += A[i][k] * B[k][j];
```

The optimized scalar version is provided in `$LAB4ROOT/benchmarks/dgemm`. Note how loop unrolling and blocking are used to improve the sequential code. Also, we should handle remainders due to loop unrolling at the end.

To measure its performance, run:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make dgemm.riscv.out
```

The goal is to vectorize the main loop and the remainder loop of the optimized code in `$LAB4ROOT/benchmarks/dgemm/{vec-dgemm-inner, vec-dgemm-remainder}.S`, respectively. For the main loop, there are blanks for vector instructions in the file. For the remainder loop, the file is almost empty, but it must be similar to and simpler than the main loop. This problem must be a good practice for the open-ended portion.

When you are ready to test your code, run it on the ISA simulator

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make vec-dgemm.riscv.out
```

When your executes correctly, collect your results from the simulations and fill out the corresponding entries in Table 1.

In your report, provide only snippet of code *you* wrote, explain the performance difference between the scalar and your vector implementation.

3.6 Vectorizing Complex Multiply (`complxmult`)

For Problem 3.6, you will vectorize Complex Multiply (`cmplxmult`). Complex multiply involves multiplying two vectors of complex numbers together element-wise. The pseudo-code is shown below:

```
1 // pseudo code
2   for ( i = 0; i < n; i++ )
3   {
4     e = (a*b) - (c*d);
5     f = (c*b) + (a*d);
6   }
```

In terms of calculating FLOPs, each iteration involves four FP multiplies and two FP adds, for a total of six FLOPs per iteration. The actual C code, found in `$LAB4ROOT/benchmarks/cmplxmult`, is shown here:

```
1  struct Complex
2  {
3    float real;
4    float imag;
5  };
6
7  // scalar C implementation
8  void cmplxmult( int n, struct Complex a[], struct Complex b[],
       struct Complex c[] )
9  {
10   int i;
11   for ( i = 0; i < n; i++ )
12   {
13     c[i].real = (a[i].real * b[i].real) - (a[i].imag * b[i].imag);
14     c[i].imag = (a[i].imag * b[i].real) + (a[i].real * b[i].imag);
15   }
16 }
```

Add your RISC-V Vector code to `$LAB4ROOT/benchmarks/vec-cmplxmult/cmplxmult.S`. You will find empty lines for vector instructions in the file, as well as a brief description of the RISC-V ABI calling convention (which provides suggestions on which registers to use).

When you are ready to test your code, much like before, run it on the ISA simulator:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make vec-cmplxmult.riscv.out
```

Also, to compare it against its scalar version, run:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make cmplxmult.riscv.out
```

Collect your results from simulations and fill out the corresponding entries in Table 1.

Hints: You will probably want to work with *strided* vector memory operations for this problem. For a strided vector load, see:

- `vlds vd, offset(rs1), rs2`.

The integer register `rs1` holds the base address of the starting address for the vector strided loads and `offset`, which is in general 0, is the offset of the starting address. The integer register `rs2` holds the size of the stride. Because this problem involves vectors of structs, and each complex number struct is 8 bytes in size, trying to load a vector of the *real* parts of the complex numbers will involve a stride value of 8 (bytes). The corresponding store version is:

- `vsts vs3, offset(rs1), rs2`.

Although not necessary, you may also get higher performance by using "fused multiply add/sub" instructions, which are also shown in CSAXPY (Line 25 in Figure 6). These instructions allow *two* vector operations to be issued in a single cycle, doubling floating point performance! Here is the list:

- `vmadd vd, vs1, vs2, vs3`
  `vd[i] := vs1[i] * vs2[i] + vs3[i]`
- `vmsub vd, vs1, vs2, vs3`
  `vd[i] := vs1[i] * vs2[i] - vs3[i]`
- `vnmadd vd, vs1, vs2, vs3`
  `vd[i] := -(vs1[i] * vs2[i] + vs3[i])`
- `vnmsub vd, vs1, vs2, vs3`
  `vd[i] := -(vs1[i] * vs2[i] - vs3[i])`

In your report, provide only snippet of code *you* wrote, explain the performance difference between the scalar and your vector implementation.

3.7 Vectorizing Index of MAX (`imax`)

For Problem 3.7, you will vectorize a non traditional vector application, `imax`, which finds the index of the max. The pseudo-code is as follows:

```
1 // pseudo code
2 idx = 0, max = 0.0;
3 for (i = 0 ; i < n ; i++)
4 {
5   if (l[i] > max) {
6     max = l[i];
7     idx = i;
8   }
9 }
```

At a first glance, this looks fairly innocuous. Its scalar, floating-point implementation is provided in `$LAB4ROOT/benchmarks/imax`. Figure out how floating-point comparisons and fmax/fmin are used from its disassembly (`imax.riscv.dump`). Measure its performance with the following command:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make imax.riscv.out
```

Fill out `TODO`s in `$LAB4ROOT/benchmarks/vec-imax/imax.S`. When ready to test your code, run:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make vec-imax.riscv.out
```

Collect your results from simulations and fill out the corresponding entries in Table 1.

Hints: Despite the simplicity of the scalar implementation, vectorizing `imax` is not trivial. We suggest you do the following:

1. Keep the current max value to a *scalar vector register*, which is initialized zero. (For its explanation, refer to Appendix B.) Also, keep the current index in a scalar register whose initial value is zero.

2. Load a vector and find its max with reduction. You may want to use `vslide` and `vmax`.
3. Find the max among the current max and the result from Step 2.
4. Set the element whose value is equal to the new max from Step 3.
5. Find the index of the max in the vector using:
   ```
   vmfirst rd, vs1
   (rd) := ([i for i in range(0, VL) if LSB(vs1[i]) == 1] + [-1])[0]
   ```
   Thus, `vmfirst` finds the index of the first non-zero LSB element in a vector if any, but returns -1 otherwise.
6. Update the global index if necessary.

In your report, provide only snippet of code *you* wrote, explain the performance difference between the scalar and your vector implementation.

4 Open-ended Portion (7 points + 2 extra points)

For this lab, there are two open-ended questions. Each open-ended group (one, two, or three students) should select one. These are contest-based questions: the group providing the highest performance implementation of their selected open-ended problem will receive two extra points, runners-up will receive one. Groups are free to try both, but will only be eligible for bonus points on the problem they select (and write about in their report).

4.1 Learning Objectives; Rubric; Submission

In writing your optimized implementation and describing that implementation your report, we want you to demonstrate your understanding of vector architectures, the sources of their performance advantages, and how these qualities can be employed in important kernels.

We provide the following coarse rubric to help guide your effort:

1. **(1 point)** Report: provide your implementation, explain roughly how it works, and report its performance.
2. **(3 points)** Report: explain how you arrived at your implementation, calling out the *three* largest optimizations you made. How much improvement did they yeild over the previous iteration of your code? Why? If you can't call out three optimizations, for half-credit, replace each one with an example of an optimization you tried that wasn't as effective as you expected. Why didn't it improve your implementation's performance?
3. **(-1 points)** Report: length exceeds 10 pages.
4. **(1 point)** Implementation: awarded for a correct implementation that outperforms the scalar code.
5. **(2 point)** Implementation: awarded based on the performance of your implementation versus the TA's partially optimized implementation. You'll receive two points for beating it, and if not, one point for coming within 50% of it.

You will submit your report digitally over Gradescope. We will provide a means to submit your code for the competition later this week. On the cover of your report, please provide your `inst` usernames (ex. mine is `cs152-tab`).

4.2 Contest: Vectorizing and Optimizing Sparse Matrix-Vector multiply (SpMV)

For this problem, you will implement a RISC-V Vector version of sparse matrix-vector multiply (SpMV), which is extensively used for graph processing and machine learning. Unlike other dense linear algebra algorithms, most elements of the matrix are zero, which are condensed in the memory. SpMV computes $y = Ax$, where A is sparse while x is dense. Its unoptimized pseudo-code is as follows:

```
1 // pseudo code
2  for ( i <- 0 until (p.size - 1) )
3  {
4    y[i] = 0.0;
5    for ( k <- p(i) until p(i+1)) {
6      y[i] += A[k] * x[idx[k]];
7    }
8  }
```

A scalar implementation written in C can be found in `${LAB4ROOT}/benchmarks/spmv/spmv_main.c`. To measure its performance, run:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make spmv.riscv.out
```

Add your own vector implementation in `${LAB4ROOT}/benchmarks/ vec-spmv/vec-spmv.S`. When you are ready to test your code, run it on the ISA simulator:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make vec-spmv.riscv.out
```

Once your code is correct, do your best to optimize `spmv`. The team whose implementation executes correctly in the fewest number of cycles will receive two bonus points, and the runners-up will receive one.

You are only allowed to write code in the `vec-spmv` function (i.e., do *not* change any code in the `vec-spmv.c` file). If you would like to do some transformation on the inputs please only do this after you have done the non-transformed version.

Collect your results from simulations and fill out the corresponding entries in Table 1.

SpMV Hints

Common techniques that generally work well are loop unrolling, lifting loads out of inner loops and scheduling them earlier, blocking the code to utilize the full register file, transposing matrices to achieve unit-stride accesses to make full use of the data cache lines, and loop interchange.

More specific to vector processors, try and have all element loads be re-factored into vector loads. Use fused multiply-add instructions as often as possible. Also, carefully choose *which* loop(s) you decide to vectorize for this problem: not all loops can be safely vectorized!

4.3 Contest: Vectorizing and Optimizing Radix Sort (`rsort`))

For this problem, you will implement a RISC-V Vector version of Radix Sort (`rsort`), which is a non-comparative sorting algorithm. In iteration i, each element is assigned to a bucket by its i'th digit from LSB. After all elements are allocated to buckets, they are merged sequentially to the list. This algorithm repeats until all the digits in every element are looked up. Its unoptimized pseudo-code is shown below:

```
1  // pseudo code
2  // MAXVAL: the max value of the type
3   while ( BASE ** i <= MAXVAL ) {
4     // Number of buckets = BASE
5     for (k = 0 ; k < BASE ; k++) {
6       buckets[k] = [];
7     }
8     for (j = 0 ; j < size(array) ; j++) {
9      number = array[j];
10     // Compute the i'th digit from LSB
11     digit = (number / (BASE ** i)) % BASE;
12     // Assign number to a bucket
13     buckets[digit].append(number);
14    }
15    // Merge buckets sequentially
16    new_array = []
17    for (k = 0 ; k < BASE ; k++) {
18      new_array += bucket[k];
19    }
20    array = new_array;
21    i++;
22  }
```

A scalar implementation written in C can be found in `${LAB4ROOT}/benchmarks/ rsort/rsort.c` To measure its performance, run:

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make rsort.riscv.out
```

When you are ready to test your code, run it on the ISA simulator

```
inst$ cd ${LAB4ROOT}/benchmarks
inst$ make vec-rsort.riscv.out
```

Once your code is correct, do your best to optimize `rsort`. The team whose implementation executes correctly in the fewest number of cycles will receive two bonus points, and the runners-up will receive one.

You are only allowed to write code in the `vec_rsort` function (i.e., do *not* change any code in the `vec-rsort.c` file). If you would like to do some transformation on the inputs please only do this after you have done the non-transformed version.

Collect your results from simulations. There are no FLOPs with `rsort`. so fill out CPIs only in Table 1. In your report, describe what your code does, and some of the strategies that you tried.

Radix Sort Hints

You can feel that getting correctly-working code is very challenging. Therefore, do not consider performance optimization when you just kicked it off. You will heavily use gather/scatter as well as vector masks for this benchmark. Be careful when you update buckets as we may read the same bucket multiple times in a single vector! Thus, you should do a correct reduction on each vector. You will rely on not only `vslide`, `vmfirst`, but also:

- `vmpop rd, vs1`
  `(rd) := len([i for i in range(0, VL) if LSB(vs1[i]) == 1])`

General tips for SpMV can also be useful for `rsort`. Your code may have `vrem`, which burns a lot of CPU cycles (10 cycles in this lab), so use another instruction. You may also notice that all buckets can fit in a single vector, so can you keep buckets in a vector across the iteration?

5 The Third Portion: Feedback

As usual, we'd like your feedback. Please fill out the survey at `https://forms.gle/kn8HXnym5ajgmA8B8`.

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

This lab was originally designed by Donggyu Kim in spring 2018. It is heavily inspired by the previous sets of CS 152 labs (which targeted the Hwacha vector processor [1]) developed by Christopher Celio. This lab was also made possible through the work of Colin Schmidt in helping make the RISC-V tool-chain available for the RISC-V Vector ISA.

A Appendix: Debugging

Debugging your vector code can be difficult. To make matters worse, you do not have an OS to call upon, `gdb`, or `printf`. However, there are a couple of strategies that will help.

First, some simple printing functions are provided: `printstr()` and `printhex()`. These functions, found in `${LAB4ROOT}/riscv-asmtests-bmarks/riscv-bmarks/stuff/syscalls.cc`, allow you to print out a static string and an integer value respectively. This can allow you to check conditions and print out the appropriate strings from your code. This is an example to print the value v0[1]:

```
# preserve ra, a0-a3
addi sp, sp, -40
sd ra, 32(sp)
sd a0, 24(sp)
sd a1, 16(sp)
sd a2, 8(sp)
sd a3, 0(sp)
```

```
# get v0[1] and convert it to int
# a0 = argument for printhex
li a0, 1
vextract a0, v0, a0
# mv a single/double precision value to a fp register
fmv.[s|d].x f0, a0
# convert a single/double value to an integer value
fcvt.l.[s|d] a0, f0
# call printhex
jal printhex
# recover ra, a0-a3
ld ra, 32(sp)
ld a0, 24(sp)
ld a1, 16(sp)
ld a2, 8(sp)
ld a3, 0(sp)
addi sp, sp, 40
```

Second, the ISA simulator prints out an instruction trace found in `${LAB4ROOT}/benchmarks/*.riscv.out`, from which, you can figure out what values each vector instruction writes to vector registers.

The objdump of the RISC-V binaries can be found in `${LAB4ROOT}/benchmarks/*.riscv.dump`, which can be very useful for comparing with the instruction traces and verifying that the code you wrote was correctly translated by the compiler.

Finally, it can be very helpful to use a small set of data for debugging. Use a test input for each benchmark, or generate it using the scripts. However, make sure your code can run with `dataset1.h`.

B Appendix: RISC-V Vector Instructions

Table 2 the list of currently available vector instructions. For vector masks, `vm`, which is optional, can be one of:

- `v1t`: update `vd[i]` if `LSB(v1[i]) == 1`.
- `v1f`: update `vd[i]` if `LSB(v1[i]) == 0`.

Note that only `v1` can be a mask register.

Vector vs. Scalar?

It may be confusing a vector register can be configured as either vector or scalar. Roughly speaking, the difference is all elements of a vector register configured as scalar have the same value.

Unlike traditional vector ISAs, there is no instruction to load a scalar value into a vector register in the RISC-V Vector ISA. Instead, this can be done by inserting a scalar value to a vector register configured as scalar. For example, `vinsert v0, x1, x0` writes the value of `x1` to `v0[0]`. If `v0` is scalar, all elements should have the same value, and thus, all elements of `v0` will have the value of `x1`.

Instruction	Operation
`vld vd, offset(rs1), vm`	`vd[i] := mem[(rs1) + offset + i]`
`vst vs3, offset(rs1), vm`	`mem[(rs1) + offset + i] := vs3[i]`
`vlds vd, offset(rs1), rs2, vm`	`vd[i] := mem[(rs1) + offset + i * rs2]`
`vsts vs3, offset(rs1), rs2, vm`	`mem[rs1 + offset + i * (rs2)] := vs3[i]`
`vldx vd, offset(rs1), vs2, vm`	`vd[i] := mem[(rs1) + offset + vs2[i]]`
`vstx vs3, offset(rs1), vs2, vm`	`mem[(rs1) + offset + vs2[i]] := vs3[i]`
`vadd vd, vs1, vs2, vm`	`vd[i] := vs1[i] + vs2[i]`
`vsub vd, vs1, vs2, vm`	`vd[i] := vs1[i] - vs2[i]`
`vmul vd, vs1, vs2, vm`	`vd[i] := vs1[i] * vs2[i]`
`vdiv vd, vs1, vs2, vm`	`vd[i] := vs1[i] / vs2[i]`
`vrem vd, vs1, vs2, vm`	`vd[i] := vs1[i] % vs2[i]`
`vmax vd, vs1, vs2, vm`	`vd[i] := max(vs1[i], vs2[i])`
`vmin vd, vs1, vs2, vm`	`vd[i] := min(vs1[i], vs2[i])`
`vsl vd, vs1, vs2, vm`	`vd[i] := vs1[i] << vs2[i]`
`vsr vd, vs1, vs2, vm`	`vd[i] := vs1[i] >> vs2[i]`
`vseq vd, vs1, vs2, vm`	`vd[i] := vs1[i] == vs2[i] ? 1 : 0`
`vsne vd, vs1, vs2, vm`	`vd[i] := vs1[i] != vs2[i] ? 1 : 0`
`vslt vd, vs1, vs2, vm`	`vd[i] := vs1[i] < vs2[i] ? 1 : 0`
`vsge vd, vs1, vs2, vm`	`vd[i] := vs1[i] >= vs2[i] ? 1 : 0`
`vaddi vd, vs1, imm, vm`	`vd[i] := vs1[i] + imm`
`vsli vd, vs1, imm, vm`	`vd[i] := vs1[i] << imm`
`vsri vd, vs1, imm, vm`	`vd[i] := vs1[i] >> imm`
`vmadd vd, vs1, vs2, vs3, vm`	`vd[i] := vs1[i] * vs2[i] + vs3[i]`
`vmsub vd, vs1, vs2, vs3, vm`	`vd[i] := vs1[i] * vs2[i] - vs3[i]`
`vnmadd vd, vs1, vs2, vs3, vm`	`vd[i] := -(vs1[i] * vs2[i] + vs3[i])`
`vnmsub vd, vs1, vs2, vs3, vm`	`vd[i] := -(vs1[i] * vs2[i] - vs3[i])`
`vslide vd, vs1, rs2, vm`	`vd[i] := 0 ≤ (rs2) + i < VL ? vs1[(rs2) + i] : 0`
`vinsert vd, vs1, rs2, vm`	`vd[(rs2)] := (rs1)`
`vextract rd, vs1, rs2, vm`	`(rd) := vs1[(rs2)]`
`vmfirst rd, vs1`	`(rd) := ([i for i in range(0, VL)` `if LSB(vs1[i]) == 1] + [-1])[0]`
`vmpop rd, vs1`	`(rd) := len([i for i in range(0, VL)` `if LSB(vs1[i]) == 1])`
`vselect vd, vs1, vs2, vm`	`vd[i] := vs2[i] < VL ? vs1[vs2[i]] : 0`
`vmerge vd, vs1, vs2, vm`	`vd[i] := LSB(vm[i]) ? vs2[i] : vs1[i]`

Table 2: RISC-V Vector Instructions

References

[1] Y. Lee. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. PhD thesis, EECS Department, University of California, Berkeley, May 2016.