CS 152 Laboratory Exercise 4

Professor: Christopher Fletcher
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley

March 28th, 2024

Revision History

Revision	Date	Author(s)	Description
1.0	2022-04-08	hngenc	Initial release
1.2	2024-03-28	Joshua You	sp24 update

1 Introduction and Goals

In this lab, you will write RISC-V vector assembly code to gain a better understanding of how data-parallel code maps to vector-style processors, and to practice optimizing vector code for a given implementation.

While students are encouraged to discuss solutions to the lab assignments with each other, you must complete the directed portion of the lab yourself and submit your own work for these problems.

1.1 Graded Items

All code and reports are to be submitted through **Gradescope**. Please label each section of the results clearly. All directed items need to be turned in for evaluation. Your group only needs to submit *one* of the problems in the open-ended section.

- (Directed) Problem 3.4: `cmplxmult` code
- (Directed) Problem 3.5: `dgemv` code
- (Directed) Problem 3.6: `dgemm` code
- (Directed) Problem 3.7: `imax` code
- (Open-ended) Problem 4.1: `spmv` code
- (Open-ended) Problem 4.2: `rsort` code
- (Directed) Problem 5: Feedback

! → Lab reports must be written in *readable* English; avoid raw dumps of logfiles. **Your lab report must be typed, and the open-ended portion must not exceed ten (10) pages.** Charts, tables, and figures – where appropriate – are excellent ways to succinctly summarize your data.

2 Background

The RISC-V vector ISA programming model is best explained by contrast with other, popular data-parallel programming models. As a running example, we use a conditionalized SAXPY kernel, **CSAXPY**. Listings 1 and 2 show CSAXPY expressed in C as both a vectorizable loop and as a SPMD (single-program multiple-data) kernel. CSAXPY takes as input an array of boolean conditions, a scalar `a`, and vectors `x` and `y`; it computes `y += ax` for the elements for which the condition is true.

Listing 1: CSAXPY vectorizable loop

```
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 }
```

Listing 2: CSAXPY SPMD kernel with 1-D thread launch

```
1 csaxpy_spmd<<<((n-1)/32+1)*32>>>;
2
3 void csaxpy_spmd(size_t n, bool cond[], float a, float x[], float y[])
4 {
5     if (tid.x < n)
6         if (cond[tid.x])
7             y[tid.x] = (a * x[tid.x]) + y[tid.x];
8 }
```

2.1 Packed-SIMD Programming Model

Listing 3 shows CSAXPY 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 *broadcasting* (or "splatting") copies of `a` to each field of a SIMD register.

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 (lines 8–11), which correspond to the body of the `if` statement in Listing 1, are masked by the predicate

Listing 3: CSAXPY mapped to packed SIMD assembly with predication
In all pseudo-assembly examples presented in this section, register `a0` holds variable `n`, `a1` holds pointer `cond`, `fa0` holds scalar `a`, `a2` holds pointer `x`, and `a3` holds pointer `y`.

```
1  csaxpy_simd:
2       slli    a0, a0, 2
3       add     a0, a0, a3
4       vsplat4 vv0, fa0
5  stripmine_loop:
6       vlb4    vv1, (a1)
7       vcmpez4 vp0, vv1
8  !vp0 vlw4    vv1, (a2)
9  !vp0 vlw4    vv2, (a3)
10 !vp0 vfmadd4 vv1, vv0, vv1, vv2
11 !vp0 vsw4    vv1, (a3)
12      addi    a1, a1, 4
13      addi    a2, a2, 16
14      addi    a3, a3, 16
15      bltu    a2, a0, stripmine_loop
16      # handle fringe cases when (n % 4) != 0
17      # ...
18      ret
```

register.[1] Finally, the address registers are incremented by the SIMD width (lines 13–14), and the stripmine loop is repeated until the computation is finished (line 15) – almost.

Since the stripmine loop handles four elements at a time, extra code is needed to handle up to three *fringe* elements at the end. 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`.

Listing 4 shows CSAXPY mapped to a similar packed-SIMD architecture without predication support. The compare instruction writes the mask to a SIMD register instead (line 7). The bulk of the computation is done regardless of the condition (lines 8–10), and `vblend4` selects the new value or the old value depending on the mask (line 11). The fringe case must be handled in scalar code due to the lack of predication.

The most important drawback to packed-SIMD architectures lurks in the assembly code: The SIMD width is expressly encoded in the instruction opcodes and address generation 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 without being recompiled to use the new instructions. Conversely, code compiled for wider SIMD

[1] We treat packed-SIMD architectures generously by assuming the support of full predication. This feature was quite uncommon. Intel AVX, for example, only introduced predication with the recent AVX-512 extension in 2016, first in a subset of Xeon models and only later available in mainstream products starting with Cannon Lake.

Listing 4: CSAXPY mapped to packed SIMD assembly without predication
The `vblend4 d,m,s,t` instruction implements a select function: `d = m ? s : t`

```
1  csaxpy_simd:
2       slli    a0, a0, 2
3       add     a0, a0, a3
4       vsplat4 vv0, fa0
5  stripmine_loop:
6       vlb4    vv1, (a1)
7       vcmpez4 vv3, vv1
8       vlw4    vv1, (a2)
9       vlw4    vv2, (a3)
10      vfmadd4 vv1, vv0, vv1, vv2
11      vblend4 vv1, vv3, vv1, vv2
12      vsw4    vv1, (a3)
13      addi    a1, a1, 4
14      addi    a2, a2, 16
15      addi    a3, a3, 16
16      bltu    a2, a0, stripmine_loop
17      # handle fringe cases when (n % 4) != 0
18      # ...
19      ret
```

registers fails to execute on older machines with narrower ones.

2.2 SIMT Programming Model

Listing 5 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. The microarchitecture fetches an instruction once but then executes it on many threads simultaneously using parallel datapaths; therefore, a scalar instruction shown in the code executes like a vector instruction.

One inefficiency of this approach is immediately evident: Since the number of launched threads must be a multiple of the warp size (32 for NVIDIA GPUs), the first action taken by each thread is to determine whether it is within bounds (lines 2–3). 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, amortizes this work over the SIMD width.) Memory coalescing logic is then needed to recover the original unit-stride access pattern from the individual memory requests issued by each thread, which otherwise must be treated as a scatter/gather. 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]

[2] More recent GPU architectures, such as the RDNA ISA from AMD, have incorporated support for scalar values and vector memory operations.

Listing 5: CSAXPY mapped to SIMT assembly

```
1  csaxpy_simt:
2      mv      t0, tid
3      bgeu    t0, a0, skip
4      add     t1, a1, t0
5      lbu     t1, (t1)
6      beqz    t1, skip
7      slli    t0, t0, 2
8      add     a2, a2, t0
9      add     a3, a3, t0
10     flw     ft0, (a2)
11     flw     ft1, (a3)
12     fmadd.s ft0, fa0, ft0, ft1
13     fsw     ft0, (a3)
14 skip:
15     stop
```

2.3 Traditional Vector 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.[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 each vector instruction, up to the hardware vector length (HVL). Software manipulates the VLR by requesting a certain application vector length (AVL) with the `vsetvl` instruction; in response, the vector unit sets the active vector lenth to the shorter 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 Listing 6 shows, the difference lies in the adjustment of the VLR at the head of every loop iteration (line 3). Most importantly, the software is agnostic to the underlying hardware vector length: The same code executes correctly and with maximal efficiency on machines with any HVL. Secondly, no fringe code is required at all: On the final trip through the loop, the VLR is simply set to the exact remainder.

The advantages of traditional vector architectures over the SIMT approach are owed to the coupled scalar control processor. The scalar register file holds only one copy of the array pointers and 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.

Listing 6: CSAXPY mapped to traditional vector assembly

```
1  csaxpy_tvec:
2  stripmine_loop:
3       vsetvl   t0, a0
4       vlb      vv0, (a1)
5       vcmpez   vp0, vv0
6  !vp0 vlw      vv0, (a2)
7  !vp0 vlw      vv1, (a3)
8  !vp0 vfmadd.s vv0, vv0, fa0, vv1
9  !vp0 vsw      vv0, (a3)
10      add      a1, a1, t0
11      slli     t1, t0, 2
12      add      a2, a2, t1
13      add      a3, a3, t1
14      sub      a0, a0, t0
15      bnez     a0, stripmine_loop
16      ret
```

2.4 RISC-V Vector Programming Model

The RISC-V "V" vector extension (RVV) resembles a traditional vector ISA, with some key distinctions. In particular, the organization of the vector register file is dynamically configurable through the *vector type register* (`vtype`), which consists of two fields: SEW and LMUL. These dictate how the vector state is conceptually partitioned along two orthogonal dimensions.

The *standard element width* (SEW) sets the default width of each vector element in bits. A narrower SEW enables elements to be more densely packed into the available storage, increasing the maximum vector length. SEW also determines the operation widths of *polymorphic* vector instructions, which allow reusing the same instruction to support a variety of data widths (e.g., both single-precision and double-precision floating-point), thereby conserving valuable opcode space.

The *length multiplier* (LMUL) is an integer power of 2 (from 1 to 8) that controls how many consecutive vector registers are grouped together to form longer vectors. With LMUL=2, vector registers vn and v$n+1$ are operated on as one vector with twice the maximum vector length. Instructions must use vector register specifiers evenly divisible by LMUL; attempts to invalid specifiers raise an illegal instruction exception. LMUL serves to increase efficiency through longer vectors when fewer architectural registers are needed, as well as to accommodate mixed-width operations (as a mechanism to maintain identical vector lengths among vectors of different datatype widths).

Listing 7 shows CSAPY mapped to RVV 0.10. The `vsetvli` instruction enables both the vector length and `vtype` to be configured in one step. SEW is initially set to 8 bits (line 3) to load the boolean values from `cond`. The second `vsetvli` instruction (line 6) widens SEW to 32 for single-precision operations; the special use of `x0` for the requested vector length has the effect of retaining the current vector length.

Listing 7: CSAXPY mapped to RISC-V V assembly

```
1  csaxpy_rvv:
2  stripmine_loop:
3      vsetvli t0, a0, e8, m2, ta, ma  # set VL; configure SEW=8 LMUL=2
4      vle8.v v8, (a1)                 # load cond[i]
5      vmsne.vi v0, v8, 0              # set mask if cond[i] != 0
6      vsetvli x0, x0, e32, m8, ta, mu # configure SEW=32 LMUL=8; retain VL
7      vle32.v v8, (a2)                # load x[i]
8      vle32.v v16, (a3)               # load y[i]
9      vfmacc.vf v16, fa0, v8, v0.t    # y[i] = (a * x[i]) + y[i] if cond[i] != 0
10     vse32.v v16, (a3)               # store y[i]
11     sub a0, a0, t0                  # decrement n
12     add a1, a1, t0                  # bump pointer cond
13     slli t0, t0, 2                  # scale VL to byte offset
14     add a2, a2, t0                  # bump pointer x
15     add a3, a3, t0                  # bump pointer y
16     bnez a0, stripmine_loop
17     ret
```

There are no separate vector predicate registers in RVV, which reduces the minimum architectural state. In the base V extension, predicated instructions always use `v0` as the source of the vector mask, while other vector registers can be used to temporarily hold mask values computed with vector logical and comparison instructions. Annotating a maskable instruction with `v0.t` (line 9) causes the operation to be executed conditionally based on the least-significant bit of the mask element in `v0`.

In the example, note that the vector mask is computed under SEW=8 but consumed under SEW=32. For the predicate bit of each element to remain in the same positions under both `vtype` settings, the SEW/LMUL ratio must be kept constant. (The "Mask Register Layout" section explains the constraints involved.) Hence, it is necessary to set LMUL=2 when SEW=8 to match the use of LMUL=8 later.

2.4.1 Specification Versioning

! → Be sure to use the correct version of the RVV specification for this lab.

The V extension has evolved substantially over the past few years. Both this lab and the lecture use the 0.10 draft of the specification, archived at `https://inst.eecs.berkeley.edu/~cs152/sp22/handouts/sp22/riscv-v-spec-0.10.html`.

The latest working draft is available at `https://github.com/riscv/riscv-v-spec`. In particular, the vector extension was officially ratified in November 2021, and the first commercial processors implementing it have become available since then. Older iterations of Lab 4 were built on the 0.4 specification, from which the current proposal diverges radically enough that they should be regarded as two different ISAs.

3 Directed Portion (50%)

This lab focuses on writing programs that target the RISC-V vector ISA. This will involve:

1. Writing vector assembly code for different benchmarks
2. Testing their correctness and estimating performance using the RISC-V ISA simulator, `spike`

Although normally just a functional simulator, to create a more interesting lab, `spike` has been extended with a rudimentary timing model of a single-issue in-order core with a standard vector unit. This timing model approximates instruction latencies from data and structural hazards, multi-cycle functional units, cache misses, and branch mispredictions.

For these simulations, `spike` is configured to model the following hardware parameters, intended to closely match the default RocketConfig design from Chipyard.

Vector unit:

- 512-bit hardware vector length (VLEN), 128-bit datapath
- Two vector functional units (VFU0, VFU1)
  - Both VFUs contain an integer ALU, an integer multiplier, and floating-point FMA units
  - VFU0 handles reductions
  - VFU1 handles floating-point conversions and comparisons
- One vector memory unit (VMU)
  - 128-bit interface to L1 data cache
  - 128 bits/cycle bandwidth for unit-stride memory operations
  - 1 element/cycle bandwidth for constant-stride and indexed memory operations (including vector AMOs)
- In-order issue with flexible vector chaining

Functional units:

- Integer ALU, 1-cycle latency
- Unpipelined scalar integer multiplier (8 bits/cycle) and divider (1 bit/cycle)
- Pipelined vector integer multiplier, 3-cycle latency
- Pipelined scalar and vector floating-point units
  - 4-cycle double-precision FMA latency
  - 3-cycle single-precision and half-precision FMA latency
  - 2-cycle floating-point conversion and comparison latency
- Unpipelined vector reduction unit, 1 element/cycle

Memory hierarchy:

- L1 instruction cache: 16 KiB, 4-way, 64 B lines
- Blocking L1 data cache: 16 KiB, 4-way, 64 B lines, 2-cycle latency for scalar loads
- Inclusive L2 cache: 512 KiB, 8-way, 64 B lines, 20-cycle latency
- 75-cycle main memory latency[3]

[3] Equivalent to Rocket Chip with a 1 GHz mbus frequency and 667 MHz DRAMSim2 frequency

Branch prediction:

- 3-cycle branch misprediction penalty
- 28-entry fully-associative BTB
- 512-entry BHT, gshare with 8 bits of global history
- 6-entry RAS

3.1 Setup

To complete this lab, `ssh` into an instructional server with the instructional computing account provided to you. The lab infrastructure has been set up to run on the `eda{1..16}.eecs.berkeley.edu` machines (`eda-1.eecs`, `eda-2.eecs`, etc.). UNIX shell commands to be run on the host are prefixed with the prompt "`eecs$`".

Once logged in, source the following script to initialize your shell environment so as to be able to access to the tools for this lab.[4] Run it before each session.

```
eecs$ source ~cs152/sp24/cs152.lab4.bashrc
```

First, clone the lab materials into an appropriate workspace and initialize the submodules.

```
eecs$ git clone https://github.com/ucb-bar/chipyard-cs152-sp24.git \
-b cs152-lab4-sp24 lab4
eecs$ cd lab4
eecs$ LAB4ROOT="$(pwd)"
```

This document henceforth uses `${LAB4ROOT}` to denote the path of the `lab4` working tree.

3.2 Register Convention

When writing assembly code, strictly adhere to the integer and floating-point register conventions set forth by the RISC-V psABI[5] (processor-specific application binary interface). Inadvertently clobbering registers will cause compatibility issues when linked with compiled code.

The `x` registers `s0`–`s11` are callee-saved, which should be preserved across function calls by saving on the stack and restoring them if used. `t0`–`t6` and `a0`–`a7` can be used as temporaries. `gp` and `tp` are reserved for special purposes in the execution environment and should be avoided. Similarly for the `f` registers, `fs0`–`fs11` are callee-saved. `ft0`–`ft11` and `fa0`–`fa7` can be used as temporaries.

Currently, all vector registers `v0`–`v31` are treated as caller-saved.

3.3 Conditional SAXPY (`csaxpy`)

The full vector code for `csaxpy` is already provided to you in `${LAB4ROOT}/benchmarks/vec-csaxpy/vec_csaxpy.S`. It is essentially identical to the example described earlier in

[4] This lab requires a different software toolchain that has experimental support for RVV 0.10.
[5] https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md

Section 2.4. Take a moment to study how it works; although relatively simple, it is a useful demonstration of some important ISA features such as SEW, LMUL, and predication.

For comparison, the scalar version is also available in `${LAB4ROOT}/benchmarks/csaxpy`.

Build and run both benchmarks on `spike` as follows:

```
eecs$ cd ${LAB4ROOT}/benchmarks
eecs$ make csaxpy.riscv.out
eecs$ make vec-csaxpy.riscv.out
```

Now that you understand the infrastructure, how to run benchmarks, and how to collect results, you can begin writing your own benchmarks.

3.4 Complex Vector Multiplication (`cmplxmult`)

`cmplxmult` multiplies two vectors of single-precision complex values. The psuedocode is shown in Listing 8.

Listing 8: `cmplxmult` pseudocode

```
1 struct Complex {
2     float real;
3     float imag;
4 };
5
6 for (i = 0; i < m; i++) {
7     c[i].real = (a[i].real * b[i].real) - (a[i].imag * b[i].imag);
8     c[i].imag = (a[i].imag * b[i].real) + (a[i].real * b[i].imag);
9 }
```

Build and run the scalar version provided in `${LAB4ROOT}/benchmarks/cmplxmult/`:

```
eecs$ make cmplxmult.riscv.out
```

Your task is to vectorize the code. Complete the assembly function in `${LAB4ROOT}/benchmarks/vec-cmplxmult/vec_cmplxmult.S` according to the TODO comments. When you are ready to test your code, build and run it on the ISA simulator:

```
eecs$ make vec-cmplxmult.riscv.out
```

If no errors are reported, you are done!

! → Refer to Appendix A for debugging tips.

3.4.1 Segmented Vector Memory Operations

When working with arrays of structs, you may want to use *segmented* vector memory operations to conveniently unpack each field into separate (consecutively numbered) vector

registers. These are described in the "Vector Load/Store Segment Instructions" section of the RVV spec. [6]

3.4.2 Fused Multiply-Add Operations

Although not necessary, more efficient code can be written by using *fused multiply-add* instructions that issue two vector floating-point operations for each instruction. These come in two destructive forms that overwrite one of the vector register operands, either the addend or the first multiplicand.[7] All relevant fused varieties are listed in the "Vector Single-Width Floating-Point Fused Multiply-Add Instructions" section.

3.5 Double-Precision Generalized Matrix-Vector Multiplication (`dgemv`)

`dgemv` performs double-precision matrix-vector multiplication $y \leftarrow Ax + y$, where x and y are vectors and A is an $m \times n$ matrix. It is a fundamental kernel part of BLAS (Basic Linear Algebra Subprograms) Level 2. Since the arithmetic intensity is relatively low (each element of A is used only once), its performance is typically memory-bound.

The unoptimized pseudocode is shown in Listing 9. The matrix `A` is stored in *row-major* order, i.e., the entries in a row are contiguous in memory.

Listing 9: Unoptimized `dgemv` pseudocode

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

Build and run the scalar version provided in `${LAB4ROOT}/benchmarks/dgemv/`:

```
eecs$ make dgemv.riscv.out
```

Your task is to vectorize the inner loop along the `n` dimension.[8] Complete the assembly function in `${LAB4ROOT}/benchmarks/vec-dgemv/vec_dgemv.S` according to the TODO comments. When you are ready to test your code, build and run it on the ISA simulator:

```
eecs$ make vec-dgemv.riscv.out
```

[6] Since their implemention can be non-trivial, the segment instructions used to be specified as an optional extension (Zvlsseg). However, they are now required for compliant RVV implementations as of v1.0.

[7] As one of the design constraints on the base V extension is to not consume too much 32-bit encoding space, non-destructive three-operand instructions are not provided.

[8] Alternatively, the `i` and `j` loops could be interchanged, which would permit the vector load of `x` to be hoisted out of the inner loop and reused for all rows of `A`. However, each iteration would necessitate a reduction operation. If the matrix were instead transposed to column-major order, vectorization along the `m` dimension would be simpler as explicit reductions can be entirely avoided while fully reusing each element of `x`.

3.5.1 Reductions and Scalars

Note that the inner product to compute `y[i]` involves a sum reduction. For long vectors particularly, reduction operations can be somewhat expensive due to the inter-element communication required. Thus, the recommended approach is to stripmine the loop to accumulate the partial sums in parallel, and then reduce the vector at the end of the loop to yield a single value.

You may find the "Vector Single-Width Floating-Point Reduction Instructions" section of the RVV spec to be useful.[9] Note that the vector reduction instructions interact with *scalar* values held in *vector* registers: The scalar result is written to element 0 of the destination vector register, and the operation also takes an initial scalar input from element 0 of a second vector operand. Refer to "Floating-Point Scalar Move Instructions" for transferring a single value between an `f` register and a vector register.

3.6 Double-Precision Generalized Matrix-Matrix Multiplication (`dgemm`)

`dgemm` performs double-precision matrix-matrix multiplication $C \leftarrow AB+C$, where A and B are both $n \times n$ matrices. This is another fundamental kernel for scientific computing and machine learning, part of BLAS Level 3. The unoptimized pseudocode is shown in Listing 10. All matrices are stored in row-major order.

Listing 10: Unoptimized `dgemm` pseudocode

```
1 for (i = 0; i < n; i++) {
2     for (j = 0; j < n; j++) {
3         for (k = 0; k < n; k++) {
4             C[i][j] += A[i][k] * B[k][j];
5         }
6     }
7 }
```

The optimized scalar version is provided in `${LAB4ROOT}/benchmarks/dgemm/`. Note how loop unrolling (of `i` and `k`) and register blocking expose opportunities for data reuse to improve efficiency. Some extra code is needed to handle the remainder after the unrolled loop. For simplicity, this implementation is not cache-blocked, although doing so would be a straightforward transformation. Build and run the scalar code as follows:

```
eecs$ make dgemm.riscv.out
```

Your task is to vectorize the second loop over `j`. Submatrices of `C` and `B` can be held in vector registers, while the entries of `A` are loaded as scalars. This naturally leads to using vector-scalar operations to compute the partial products. As a hint, first work through the matrix multiplication by hand to see how the computation can be rearranged into a vectorizable pattern.

[9] On vector architectures that do not feature explicit reduction instructions, this can be implemented by recursively halving the vector length and adding both halves together with vector addition.

Complete the assembly functions for the main inner loop in `${LAB4ROOT}/benchmarks/vec-dgemm/vec_dgemm_inner.S` and the remainder loop in `${LAB4ROOT}/benchmarks/vec-dgemm/vec_dgemm_remainder.S`. Try to leverage fused multiply-add instructions where possible. When you are ready to test your code, build and run it on the ISA simulator:

```
eecs$ make vec-dgemm.riscv.out
```

3.7 Index of Maximum Element (`imax`)

In this problem, you will vectorize a less conventional vector application, `imax`, which finds the index of the largest value in an array. One use case is for identifying the pivot element in certain matrix algorithms, such as Gaussian elimination. The pseudocode is shown in Listing 11.

Listing 11: `imax` pseudocode

```
1 idx = 0, max = -INFINITY;
2 for (i = 0; i < n; i++) {
3     if (x[i] > max) {
4         max = x[i];
5         idx = i;
6     }
7 }
```

Build and run the scalar version provided in `${LAB4ROOT}/benchmarks/imax/`:

```
eecs$ make imax.riscv.out
```

Despite the simplicity of the scalar implementation, vectorizing `imax` is not as trivial. The following approach is suggested:

1. Keep the current maximum in an `f` register, initialized to negative infinity, and the current index in an `x` register, initialized to zero.
2. Load a vector and find its maximum with a reduction.
3. Compare against the global maximum.
4. Use a vector floating-point comparison to represent the location of the maximum element as a vector mask.
5. Find the first set bit in the mask using `vfirst.m`. This yields the index of the lowest-numbered element of the mask vector that has its least-significant bit set, or -1 otherwise.
6. Update the global index and maximum if necessary.

Once you understand how the reduction and mask operations work, complete the assembly function in `${LAB4ROOT}/benchmarks/vec-imax/vec_imax.S` according to the TODO comments. When you are ready to test your code, build and run it on the ISA simulator:

```
eecs$ make vec-imax.riscv.out
```

3.8 Submission

Run the following to collect all of your code for the directed portion into one archive, and upload `directed.zip` to the Gradescope autograder.

```
eecs$ make zip-directed
```

The following source files should be present at the root of the ZIP file:

- `vec_cmplxmult.S`
- `vec_dgemv.S`
- `vec_dgemm_inner.S`
- `vec_dgemm_remainder.S`
- `vec_imax.S`

The directed problems are evaluated based on correctness, so please check that your code passes the autograder test suite.

(No written report is required for the directed portion this time.)

4 Open-Ended Portion (50%)

Select *one* of the following questions per team.

In writing your optimized implementation and describing your methodology in the report, the goal is to demonstrate your understanding of vector architectures, the sources of their performance advantages, and how these qualities can be employed in important computational kernels.

4.1 Sparse Matrix-Vector Multiplication (`spmv`)

For this problem, you will implement and optimize RISC-V vector code for sparse matrix-vector multiplication (SpMV), which is extensively used for graph processing and machine learning. SpMV computes $y = Ax$, where A is a *sparse* matrix and x is a dense vector. Unlike other dense linear algebra algorithms, most entries of A are zero, and the matrix is represented in a condensed format in memory. The unoptimized pseudocode is shown in Listing 12.

Listing 12: `spmv` pseudocode

```
1 for (i = 0; i < n; i++) {
2     for (k = ptr[i]; k < ptr[i+1]; k++) {
3         y[i] += A[k] * x[idx[k]];
4     }
5 }
```

A scalar reference implementation is provided in `${LAB4ROOT}/benchmarks/spmv`. Build and run the benchmark as follows:

```
eecs$ make spmv.riscv.out
```

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

```
eecs$ make vec-spmv.riscv.out
```

4.1.1 Optimization

Once your code is correct, do your best to optimize `spmv` to minimize the number of cycles (per `mcycle`).

You are only allowed to write code in `vec_spmv.S`; do not change any code in `vec_spmv_main.c` except for debugging. If you would like to perform some transformation on the inputs, only do so after you have verified the non-transformed version.

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

More specific to vector architectures, try to refactor all element loads into vector loads. Use fused multiply-add instructions where possible. Also, carefully choose which loop(s) to vectorize for this problem, as not all loops can be safely vectorized!

4.1.2 Submission

! → Please ensure that your code is appropriately commented. Use the following command to archive your code for submission, and record the resulting `vec-spmv.zip` file to the Gradescope autograder.

```
eecs$ make zip-spmv
```

In a separate report, roughly explain how your SpMV implementation works, and report the dynamic instruction count and cache statistics. Also explain how you arrived at your implementation, and describe at least three optimizations that you applied in detail. How much improvement did they yield over the previous iterations of your code?

4.2 Radix Sort (`rsort`)

For this problem, you will implement and optimize RISC-V vector code for radix sort (`rsort`), a non-comparative integer sorting algorithm. The unoptimized pseudocode is shown in Listing 13. In iteration i, each value is assigned to a bucket by its i-th digit, starting from the least-significant digit. After all values are allocated to buckets, they are merged sequentially into a new list. This algorithm repeats until all digits in every value have been traversed.

Listing 13: `rsort` pseudocode

```
1  // TYPE_MAX: maximum value of the data type
2  for (power = 1; power < TYPE_MAX; power *= BASE) {
3      // Number of buckets = BASE
4      for (k = 0; k < BASE; k++) {
5          buckets[k] = [];
6      }
7      for (j = 0; j < ARRAY_SIZE(array); j++) {
8          key = array[j];
9          // Extract next digit
10         digit = (key / power) % BASE;
11         // Assign value to bucket
12         buckets[digit].append(key);
13     }
14     // Merge buckets sequentially
15     new_array = [];
16     for (k = 0; k < BASE; k++) {
17         new_array.extend(bucket[k]);
18     }
19     array = new_array;
20 }
```

A scalar reference implementation is provided in `${LAB4ROOT}/benchmarks/rsort`. Rather than directly assigning the values to buckets, the optimized version uses the buckets to produce a histogram of occurrences of each digit, which is then used to compute the offset of each element in the new array. Thus, the merge step becomes a permutation. Build and run the benchmark as follows:

```
eecs$ make rsort.riscv.out
```

Add your own vector implementation to `${LAB4ROOT}/benchmarks/vec-rsort/vec_rsort.S`. The data values are 32-bit `unsigned int`s. When you are ready to test your code, build and run it on the ISA simulator:

```
eecs$ make vec-rsort.riscv.out
```

4.2.1 Optimization

You can feel that this may be a more challenging algorithm to vectorize than usual, so focus on first getting the code to work correctly before embarking on any complicated optimizations.

Once your code passes the given test, do your best to optimize `rsort` to minimize the number of cycles (per `mcycle`). You are only allowed to write code in `vec_rsort.S`; do not change any code in `vec_rsort_main.c` except for debugging.

Indexed vector memory operations (scatter/gatter) and predication will be heavily used in this benchmark. Take care when updating buckets, as the same bucket may be accessed multiple times by a vector operation.

The general suggestions for `spmv` in Problem 4.1 also apply to `rsort`. In practice, `vrem` can be expensive in terms of performance, being essentially a long-latency divide operation per element, so it should be avoided. You may also notice that every bucket might be able to fit in a single vector; if so, consider how buckets could be kept in the same vectors across iterations.

5 Feedback Portion

In order to improve the labs for the next offering of this course, we would like your feedback. Please append your feedback to your individual report for the directed portion.

- How many hours did you spend on the directed and open-ended portions?
- What did you dislike most about the lab?
- What did you like most about the lab?
- Is there anything that you would change?
- Is there something else you would like to explore in the open-ended portion?
- Are you interested in modifying hardware designs as part of the lab?

Feel free to write as much or as little as you prefer (a point will be deducted only if left completely empty).

5.1 Team Feedback

In addition to feedback on the lab itself, please answer a few questions about your team:

- In a few sentences, describe your contributions to the project.
- Describe the contribution of each of your team members.
- Do you think that every member of the team contributed fairly? If not, why?

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 developed by Christopher Celio, which targeted the Hwacha vector processor [2].

References

[1] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978. DOI: 10.1145/359327.359336.

[2] Y. Lee, "Decoupled vector-fetch architecture with a scalarizing compiler," Ph.D. dissertation, EECS Department, University of California, Berkeley, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-117.html.

A Appendix: Debugging

For each benchmark, the ISA simulator prints an instruction trace to `${LAB4ROOT}/benchmarks/*.riscv.out`.

The disassembly for specific benchmarks can be dumped as follows, which can be useful for comparing against the instruction traces and verifying that the code was assembled as expected.

```
eecs$ cd ${LAB4ROOT}/benchmarks
eecs$ make <bmark>.riscv.dump
```

It is also possible to generate a more detailed commit log that records every value written to each destination register and the address stream of memory accesses.

```
eecs$ make <bmark>.riscv.log
```

The current SEW, LMUL, and vector length are also logged for each vector instruction. Note that the contents of a vector register are shown concatenated as a single raw hex value; refer to the "Mapping of Vector Elements to Vector Register State" section in the RVV spec for how to unpack the layout. For `LMUL` > 1, the `v` registers that comprise a vector register group are displayed separately, possibly in an arbitrary order.

Finally, it can be very helpful to debug using a smaller dataset. Switch to `dataset2.h` for the open-ended problems, or generate custom input data using the provided scripts. However, make sure that your code eventually passes the test using `dataset1.h`.