C152 Laboratory Exercise 4

Professor: John Wawrzynek
TA: Martin Maas
Department of Electrical Engineering & Computer Science
University of California, Berkeley

October 30, 2016

1 Introduction and goals

The goal of this laboratory assignment is to allow you to explore the *vector-fetch* architecture using the `Chisel` simulation environment.

You will be provided a complete implementation of a *vector-fetch* style processor, called *Hwacha*. Students will write *vector-fetch* assembly code targeting *Hwacha*, 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 a vector implementation of spare-matrix vector multiply, or attempt to improve *Hwacha*'s prefetching

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.

Students are encouraged to discuss solutions to the lab assignments with other students, but must run through the directed portion of the lab by themselves and turn in their own lab report. For the open-ended portion of each lab, students will work individually or in groups of two or three. Each group will turn in a single report for the open-ended portion of the lab. Students are free to take part in different groups for different lab assignments.

For this lab, there will only be two open-ended assignments. 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 The *vector-fetch* Architecture

The *vector-fetch* architecture is a new style of data-parallel architecture that combines the efficiencies of traditional vector processors with the programability of general purpose GPU processors.[1, 2, 3]

The Hwacha assembly 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**; it computes **y** += **ax** for the elements for which the condition is true.

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

(a) vectorizable loop

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

(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**. 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, which in turn is used

```
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`.

to set a predicate register. The next four instructions, 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, and the stripmine loop is repeated until the computation is finished—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, 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

[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_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.**

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.

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 decide to do nothing. 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

```
csaxpy_tvec:
stripmine_loop:
     vsetvl  t0, a0
     vlb     vv0, (a1)
     vcmpez  vp0, vv0
!vp0 vlw     vv0, (a3)
!vp0 vlw     vv1, (a4)
!vp0 vfma    vv0, vv0, a2, vv1
!vp0 vsw     vv0, (a4)
     add     a1, a1, t0
     slli    t1, t0, 2
     add     a3, a3, t1
     add     a4, a4, t1
     sub     a0, a0, t0
     bnez    a0, stripmine_loop
     ret
```

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

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. 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 equal to the HVL.

The Programmer's View of *vector-fetch*

The Hwacha architecture builds on the traditional vector architecture, with a key difference: the vector operations have been hoisted out of the stripmine loop and placed in their own *vector fetch block*. This allows the scalar control processor to send only a program counter to the vector processor. The control processor then completes the stripmining loop faster and is able to continue doing useful work, while the vector processor is independently executing the vector instructions.

Figure 6 shows the Hwacha user-visible register state. Like the traditional vector machine, Hwacha has vector data registers (`vv`*0–255*) and vector predicate registers (`vp`*0–15*), but it also has two flavors of scalar registers. These are the *shared* registers (`vs`*0–63*, note that `vs`*0* is hardwired to constant 0), which can be read and written within a vector fetch block, and *address* registers (`va`*0–31*), which are read-only within a vector fetch block. This distinction supports non-speculative access/execute decoupling and is further described in the Hwacha microarchitecture manual.

Vector data and shared registers may hold 8-, 16-, 32-, and 64-bit integer values and half-, single-, and double-precision floating-point values. Vector predicate registers are 1-bit wide, and hold boolean values that mask vector operations. Address registers hold 64-bit pointer values, and serve as the base and stride of unit-strided and strided vector memory instructions.

In addition, the vector configuration register `vcfg`, which keeps the configuration state of the vector unit, and the vector length register `vlen`, which stores the maximum hardware vector length, are also visible to the user.

The maximum hardware vector length is configurable based on how many registers of each type a program uses. Regardless of how many registers are used, a hardware vector length of 8 is guaranteed. The vector length register (`vlen`) can be set to zero; in this case, all vector instructions will not be executed.

Figure 5 shows the CSAXPY code for the Hwacha machine. The structure of the stripmine loop in the control thread (line 1–16) is similar to the traditional vector code, however, all vector operations in the stripmine loop have been hoisted out into its own worker thread (line 18–25). The control thread first executes the `vsetcfg` instruction (line 2), which adjusts the maximum hardware vector length taking the register usage into account. `vmcs` (line 3) moves the value of a scalar register from the control thread to a `vs` register. The stripmine loop sets the vector length with a `vsetvl` instruction (line 5), moves the array pointers to the vector unit with `vmca` instructions (line 6–8), then executes a vector fetch (`vf`) instruction (line 9) causing the Hwacha unit to execute the vector fetch block. The code in the vector fetch block is equivalent to the vector code in Figure 4, with the addition of a `vstop` instruction, signifying the end of the block.

For the CSAXPY example, factoring the vector code out of the stripmine loop reduces the scalar instruction count by 15%, but as the stripmine loop gets complicated with more vector instructions, the fraction of saved scalar instruction fetches per stripmine loop due

```
csaxpy_control_thread:
  vsetcfg ...
  vmcs vs1, a2
stripmine_loop:
  vsetvl  t0, a0
  vmca    va0, a1
  vmca    va1, a3
  vmca    va2, a4
  vf      csaxpy_worker_thread
  add     a1, a1, t0
  slli    t1, t0, 2
  add     a3, a3, t1
  add     a4, a4, t1
  sub     a0, a0, t0
  bnez    a0, stripmine_loop
  ret

csaxpy_worker_thread:
     vlb     vv0, (va0)
     vcmpez  vp0, vv0
!vp0 vlw     vv0, (va1)
!vp0 vlw     vv1, (va2)
!vp0 vfma    vv0, vv0, vs1, vv1
!vp0 vsw     vv0, (va2)
     vstop
```

Figure 5: **CSAXPY kernel mapped to the Hwacha programming model.**

to the Hwacha assembly programming model will increase. This lets the control thread run ahead further, enabling a higher degree of access/execute decoupling. Consult the Hwacha microarchitecture manual for more details.

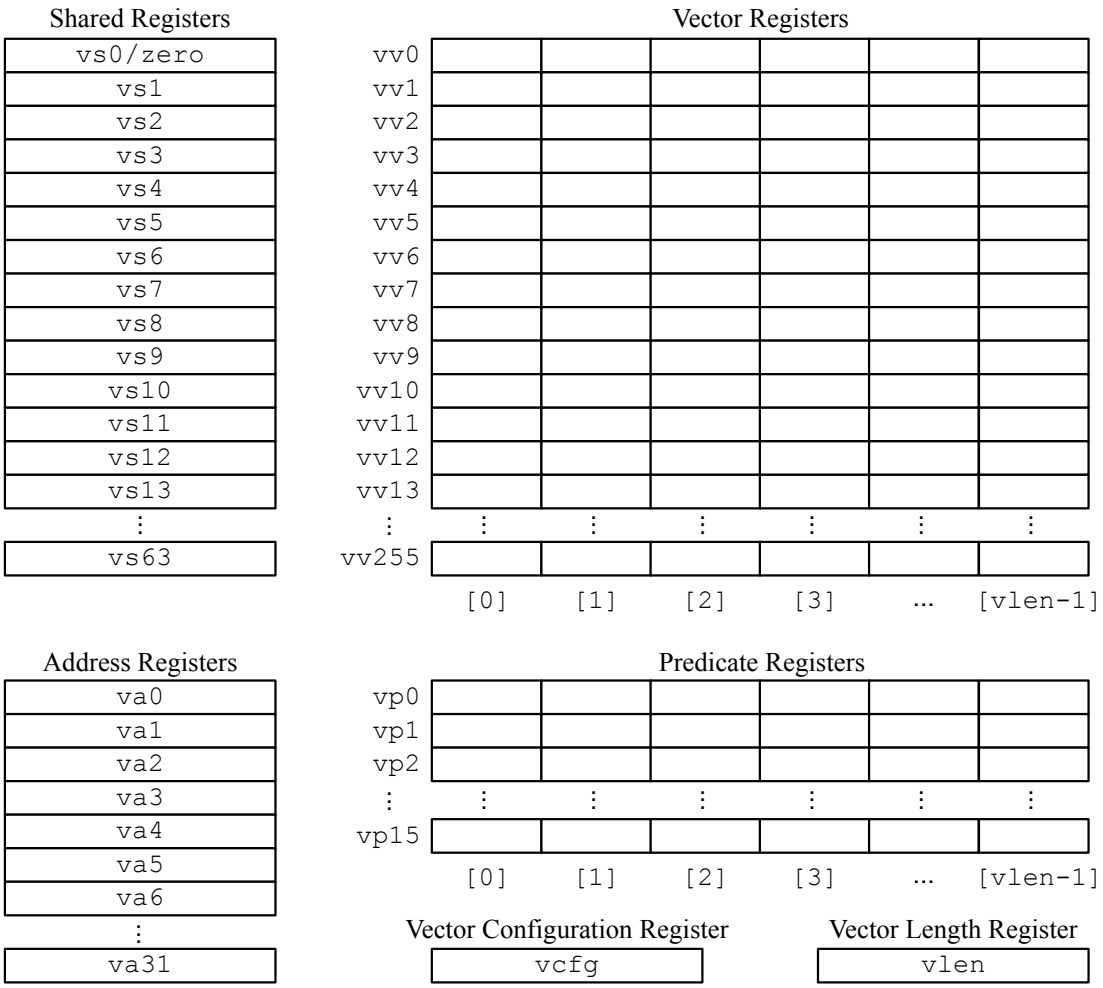

Figure 6: Hwacha user-visible register state.

2.5 The *Rocket/Hwacha vector-fetch* Processor

A `Chisel` implementation of a full *vector-fetch* processor is provided. The provided *vector-fetch* processor comes with two big pieces: the *control processor*, known as *Rocket*, and the vector unit, known as *Hwacha*.

Rocket is a RV64G 5-stage, fully bypassed in-order core. It has full supervisor support (including virtual memory). It also supports sub-word memory accesses and floating point. In short, *Rocket* supports the entire 64-bit RISC-V ISA (however, no OS will be used in this lab, so code will still run "bare metal" as in previous labs).

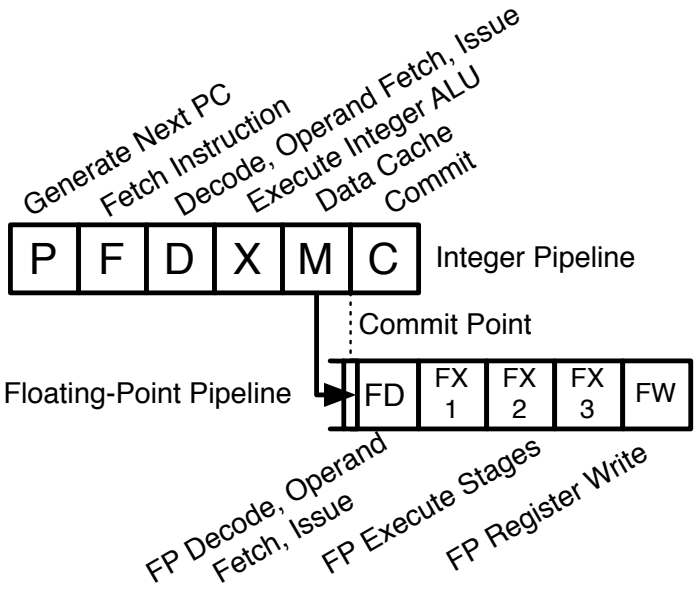

Figure 7: The *Rocket* control processor pipeline.

As the control processor, *Rocket* executes scalar code. However, when it encounters any *vector* instruction, it will send the instruction (and the corresponding operands, perhaps including a PC) to the *Hwacha* vector-unit, which will handle the command and/or begin fetching and executing instructions starting at the given PC.

Rocket has an L1 data and instruction cache while *Hwacha* has an L1 instruction cache for *vector-fetch* instructions and communicates directly with the L2 cache for data. These caches are then backed up by DRAM that lives in the test harness.

Both *Rocket* and *Hwacha* are and have been developed and debugged for many tape-outs, one of which is a joint Berkeley/MIT research chip (Berkeley is developing the cores and caches, while MIT is focusing on novel memory designs that are far beyond the scope of this lab). The upside of this is that you are playing with an actual, realistic processor design that is being used for real computer architecture research. The downside is that

many of the tools and features are not yet mature, and it can be harder to grasp all of the moving parts of these very real processors!

2.6 Graded Items

You will turn in a copy of your results via Gradescope. Please label each section of the results clearly. The directed items need to be turned in for evaluation. Your group only needs to turn in *one* of the problems found in the open-ended portion. Some of the open-ended questions also request source code – please turn those in via e-mail to the TA. This should include the files you have modified such that they can be replaced with the current versions to replicate your results.

The main result of the lab will be the following Chart 1, which compares the floating point performance of the *vector-fetch* code running on the *Hwacha* vector-unit against the reference code running on the scalar *Rocket* core (using both GFLOPs and CPI). Each problem will guide you through the steps to accomplish this task. The performance results of *Rocket* have already been completed for you.

	vvadd	saxpy	dgemm	cmplxmult	spmv
Rocket (scalar)	0.060 GFLOPs 1.94 CPI	0.080 GFLOPs 3.56 CPI	0.859 GFLOPs 1.40 CPI	0.159 GFLOPs 2.30 CPI	0.069 GFLOPs 2.76 CPI
Hwacha (vf)					

Table 1: Performance of floating point benchmarks.

1. Problem 3.3: Answer to the question at the end of the section; `vvadd, saxpy, dgemm` performance statistics and answers
2. Problem 3.4: `cmplxmult` code, statistics, and answers
3. Problem 4.1: `spmv` code, statistics, and answers
4. Problem 5: Feedback on this lab

3 Directed Portion (50% of lab grade)

3.1 General Methodology

This lab will focus on writing *vector-fetch* assembly code. This will be done in two steps: step 1) write assembly code and test it for correctness using the very fast *RISC-V ISA simulator*, and Step 2) measure the performance of your correct code on a Chisel-generated cycle-accurate simulator of the *Rocket/Hwacha* processor.

3.2 Setting Up Your `Chisel` Workspace

To complete this lab you will log in to an instructional server, which is where you will use `Chisel` and the RISC-V tool-chain. The tools for this lab were set up to run on any of the 5 instructional Linux servers `icluster5.eecs`, `icluster6.eecs`, ..., `icluster9.eecs`. (see `http://inst.eecs.berkeley.edu/cgi-bin/clients.cgi?choice=servers` for more information about available machines).

First, download the lab materials. This lab is now managed as a git repository which means you need to use git to fetch updates from the published version. To copy the repo you will need to clone it:

```
inst$ cd ~
inst$ git clone ~cs152/fa16/lab4-git lab4
inst$ cd lab4
inst$ export LAB4ROOT=$PWD
```

If any updates are released you can then pull in the new updates using

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

If you encounter problems using git feel free to post a question on Piazza or consult the git documentation (see `https://git-scm.com/doc`)

The following command will set up your bash environment, giving you access to the entire CS152 lab tool-chain. Run it before each session:[2] Note that hwacha has a different ISA and therefore needs a different tool-chain than in previous labs which could use the stock RISC-V tool-chain.

```
inst$ source ~cs152/fa16/cs152.lab4.bashrc
```

[2]Or better yet, add this command to your bash profile.

We will refer to `./lab4` as `${LAB4ROOT}` in the rest of the handout to denote the location of the Lab 4 directory. Some of the directory structure is shown below:

- `${LAB4ROOT}`/rocket-chip/
  - riscv-tools/riscv-tests/ Source code for assembly tests and benchmarks.
    - **benchmarks/** Benchmarks (mostly) written in C. This is where you will spend nearly all of your time.
      - vec-vvadd C and assembly code for the vector-vector add benchmark.
      - vec-saxpy C and assembly code for the scalar Ax plus Y benchmark.
      - vec-dgemm C and assembly code for the matrix multiply benchmark.
      - vec-cmplxmult C and assembly code for the complex-multiply benchmark.
      - vec-spmv C and assembly code for the sparse matrix vector multiply benchmark.
    - isa/ Assembly code for the individual instruction tests.
  - **emulator/** C++ simulation tools and output files.
  - csrc/ C++ test bench source code.
  - dramsim2/ DRAMSim2 source code that is used to emulate DRAM.
  - chisel/ The `Chisel` source code.
  - **hwacha/** The *Hwacha* source code.
  - rocket/ The *Rocket* processor.
  - hardfloat/ The floating point unit source code.
  - uncore/ The uncore source code.
  - src/ Top-level source code.
  - sbt/ `Chisel`/Scala voodoo. You can safely ignore this directory.

For this lab, we will play with the benchmarks `vec-vvadd`, `vec-saxpy`, `vec-dgemm`, `vec-cmplxmult`, and `vec-spmv`. To compile and run these benchmarks on the RISC-V ISA simulator, execute the following commands:

```
inst$ cd ${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks
inst$ make clean; make; make run-riscv
```

This quickly tests the benchmarks for correctness using the ISA simulator. The `vec-cmplxmult` and `vec-spmv` benchmarks should FAIL, as you have not written the code for them yet!

To run the benchmarks on the cycle-accurate C++ simulator of *Hwacha*/*Rocket*, execute the following commands:

```
inst$ cd ${LAB4ROOT}/emulator
inst$ make clean; make; make run-bmark-tests
```

If this is the first time you are compiling the emulator, this command may take a while.

The `vec-cmplxmult` and `vec-spmv` benchmarks should once again FAIL, because you have not written the code for them yet! Note that the emulator is only compiled once. Once you add your working complex multiply and sparse matrix vector multiply code, the total simulation time should be about five to ten minutes.

3.3 Measuring the Performance of Simple Kernels

To get used to the Lab 4 infrastructure and *vector-fetch* coding in general, we will first look at the provided Vector-Vector Add (`vec_vvadd`) benchmark and measure its performance on *Hwacha*. First, navigate to the `vec_vvadd` directory, found in:

```
${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/vec-vvadd/
```

In the `vec-vvadd` directory, there are a few files of interest. First, the `dataset1.h` file holds a static copy of the input vectors and results vector. Second, `vec_vvadd_main.c` holds the main driving C code for the benchmark, which includes initializing the state of the program, calling the `vvadd` function itself, and verifying the correct results of the function. An example scalar implementation of `vvadd`, written in C, is provided in `vec_vvadd.c` as well. The assembly implementations are found in `vec_vvadd_asm.S`. Two versions are provided: first, a scalar assembly version of `vvadd`, and second, a *vector-fetch* version of `vvadd`. Now let's run the vector version of `vec-vvadd` on the ISA simulator:

```
inst$ cd ${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks
inst$ make clean; make; make run-riscv
```

This will delete any old copies of the benchmarks, build new copies of the benchmarks, generate obj-dump files, generate hex file copies, and run the resulting RISC-V binaries on the RISC-V ISA simulator.[3] You should see a PASS for `vecv-vadd`, denoting that the output vector of our *vector-fetch* implementation matches the reference results provided by the `dataset.h` file. For now, you should see FAIL for both the `vec-cmplxmult` and `vec-spmv` benchmarks, since we have not yet written the code for them yet!

[3]Notice that the shown command (`make run-riscv`) runs the RISC-V binary. It is also possible to build the code and run it on the "host" x86 platform, using `make run-host`. The advantage is that you get full printf support (and a full OS), but the disadvantage is that you can not use any RISC-V assembly in your code. You should be able to ignore this option for the lab, but it might be helpful for debugging.

Now, we will run `vec-vvadd` on the cycle-accurate simulator of *Hwacha*.

```
inst$ cd ${LAB4ROOT}/rocket-chip/emulator
inst$ make clean; make; make output/vec-vvadd.riscv.out
```

You should see the following output, which corresponds to `vec-vvadd`:

```
./emulator-Top-ISCA2016Config +dramsim +max-cycles=100000000 +verbose
+loadmem=output/vec-vvadd.riscv.hex none 3>&1 1>&2 2>&3 |
/home/ff/cs152/sp16/install-lab4/bin/spike-dasm --extension=hwacha >
 output/vec-vvadd.riscv.out && [ $PIPESTATUS -eq 0 ]
cycle = 61807
instret = 574
```

The first line calls the emulator and loads the `vec-vvadd` benchmark into the simulator's memory, and stores any log information into `output/vec-vvadd.riscv.out`.

The second line is the output from the `vec-vvadd.riscv` program itself. In this example, we are provided information about the *number of cycles* executed by the critical function, and the *number of instructions retired* by the critical function in decimal form.

Use this information to calculate the CPI of the control processor (retired instructions is measured from the scalar control processor's point of view), and to calculate the FLOPs ("floating point operations / second") achieved by *Hwacha*.

To calculate the FLOPs achieved, we need to know two things: how many floating point operations were performed, and how many seconds elapsed. To calculate the former, we need to look at the `vec-vvadd` code (`vec_vvadd_main.c` and `dataset1.h`): we can see that every iteration performs one floating point add operation, and that `vec-vvadd` runs for 20000 iterations. To calculate seconds, we need to know the number of cycles that elapsed (provided by the above printout), as well as the clock rate of the processor. Both *Rocket* and *Hwacha* can run at 1 GHz. Thus, since *Rocket* is a single-issue machine, we expect its absolute maximum theoretical floating point performance to be 1 GFLOP (1 floating point op per cycle / 1 billion cycles per second).[4] *Hwacha* as a vector machine has more functional units to make consistent use of its large register files. In fact *Hwacha* has the capability to perform 4 double precision FMAs per cycle. As we have seen in the previous section, *Hwacha*'s register file is broken up into SRAM banks with 1 read and 1 write port. For microarchitectural reasons, these banks are actually 128-bits wide, so we have an aggregate read bandwidth of 8 double precision operands per cycle. **If this is the case, how can *Hwacha* fully saturate its 4 FMAs which require 3*4=12 double precision operands per cycle? Include the answer in your write-up.** (Hint: look at the csaxpy code above and think about the other register *Hwacha* can use).

[4]This is actually a bit of a lie, since *Rocket* and *Hwacha* support "fused multiply add" instructions, which perform a $d = c + (a \times b)$ operation. Thus, with the `fmadd` and `fmsub` instructions, the processor can actually issue *two* floating point operations in a single cycle!

3.4 Implementing Complex Multiply (`cmplxmult`) in *vector-fetch*

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 on the *Hwacha vector-fetch* core.

The first benchmark will be 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   }
7 }
```

In terms of calculating FLOPs, each iteration involves four FP multiplies and two FP adds, for a total of six FLOP per iteration. The actual C code 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 *vector-fetch* code to

`${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/vec_cmplxmult/vec_cmplxmult_asm.S`. You will find a brief description of the RISC-V ABI calling convention in the file (e.g., suggestions on which registers to use).

When you are ready to test your *vector-fetch* code, first test for correctness on the ISA simulator:

```
inst$ cd ${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks
inst$ make clean; make; make run-riscv
```

Once your code passes the correctness test, you can then gather performance results on cycle-accurate simulator of *Hwacha*:

```
inst$ cd ${LAB4ROOT}/rocket-chip/emulator
inst$ make clean; make; make output/vec-cmplxmult.riscv.out
```

Collect your results and fill out the corresponding entries in Table 1. Also, attach your code to your write-up.

Hints: You will almost certainly want to work with *strided* vector memory operations for this problem. For strided loads, the instruction is `vlstw vv1, rBaseAddr, rStride`, which stands for *vector load strided (word version)*. The corresponding store version is `vsstw`. The argument `vv1` is the vector register # 1 (you may use any number from 0 to the number of configured registers). The operand register `rBaseAddr` holds the starting memory address for the vector strided load to begin loading from, and `rStride` is a register that holds the size of the stride (both are required to be address registers `vaX`). 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).

Although not necessary, you may also get higher performance by using "fused multiply add" instructions, which are supported by *Hwacha* and *Rocket* ($d = c + (a \times b)$). These instructions (`vfmadd.w` and `vfmsub.w`) allow *two* floating point operations to be issued in a single cycle, doubling floating point performance! See the provided Hwacha ISA specification for more information about the provided floating point instructions (you can find it in the "Resources" section of the course website).

4 Open-ended Portion (50% of lab grade + 25% bonus)

For this lab, there will only be one open-ended portion that all students can do. As with all labs, you can work individually or together in groups of two or three.

4.1 Contest: Vectorizing and Optimizing Sparse Matrix Vector Multiply

For this problem, you will create a *vector-fetch* implementation of sparse matrix-vector multiply. A scalar implementation written in C can be found in:

```
${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/benchmarks/
vec-spmv/vec_spmv_main.c
```

Add your own *vector-fetch* implementation in `vec_spmv_asm.S`. Once your code passes the correctness test, do your best to optimize `spmv` for *Hwacha*. This will be a contest, and the best team, as measured by the achieved FLOPs (i.e., the lowest number of cycles to correctly execute), will receive bonus points equivalent to 25% of the maximum lab grade.

You are only allowed to write code in the `vec_spmv_asm` function (i.e., do *not* change any code in the `vec_spmv_main.c` file). If you would like to do some transformation on the inputs, we recommend to only do this after you have done the non-transformed version (as this will make it easier to debug; remember that only correct implementations will count towards the competition).

Email your `spmv` *vector-fetch* assembly code to the TA, and include it in your write-up as well. Describe what your code does, and some of the strategies that you tried.

Sparse Matrix Vector Multiply 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 L2 cache lines, and loop interchange.

More specific to *vector-fetch*, 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!

Finally, be mindful about the use of the `fence.v` instruction: it is expensive and can hurt performance, but you *must* use it when you need to make the results of stores visible.

5 The Third Portion: Feedback

This is a newly refreshed lab, and as such, we would like your feedback again! 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 (a point will be taken off only if left completely empty).

6 Acknowledgments

This lab was made possible through the work of Yunsup Lee and Andrew Waterman (among others) in developing the *Rocket* and *Hwacha* processors, and in helping make the RISC-V tool-chain available to users at large. This lab was originally developed for CS152 at UC Berkeley by Christopher Celio.

A Appendix: Debugging

Debugging your *vector-fetch* 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}/rocket-chip/riscv-tools/riscv-tests/benchmarks/commmon/syscalls.c`

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.

Second, the ISA simulator can be run in a debug mode that prints out an instruction trace. For example, the basic command for running `vec_vvadd` in the ISA simulator is:

```
inst$ spike vec-vvadd.riscv
```

However, adding "-d" will provide a log of the instructions executed.

```
inst$ spike -d vec_vvadd.riscv
```

The only down-side is that this only shows the instruction trace from the point of view of the *control processor*: the vector unit is effectively invisible.

In order to give a better sense of the vector instructions you can rebuild spike with the `--enable-hcommitlog` flag which causes it to print out all writes to vector registers.

The objdump of the RISC-V binaries can be found in

`${LAB4ROOT}/rocket-chip/riscv-tools/riscv-tests/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.

If you are confused about vector-thread, I recommend that you look at the Hwache manual on the course website, or the Hwacha website at `http://www.hwacha.org`.

References

[1] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *IEEE Micro*, 24(6):84–90, 2004. [`http://groups.csail.mit.edu/cag/scale/papers/vta-isca2004.pdf`].

[2] Y. Lee, 2012. Maven Slides from ISCA 2011 [`http://www.eecs.berkeley.edu/~yunsup/papers/maven-isca2011-talk.pdf`].

[3] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. June 2011.