Lab 4 Open Ended Section

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.

Spmv pseudocode

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

Design Approach

First we thought about which loops can be vectorized and how we can achieve that. We thought about vectorizing the Y values, however we determined that it wouldn't be efficient because we would be adding one partial sum at a time to the Y vector. Then we thought about vectoring the partial sums, which we think is the ideal case. This is because it is easy to implement and it also reduces the amount of cycles used up when loading and branching. After that we applied other techniques such as loop unrolling and changing the instruction order to optimize it.

Overview of Algorithm

Vec_spmv:

- Set the outer loop index at t6 to 0

Vec_spmv_loop_i:

- Get current k value at ptr[i], store in t1
- Get the next k value at ptr[i + 1] - 3, store in t2
- Zero out the vector we use to collect the partial sums

Vec_spmv_loop_k:

- Save address index + k * integer offset into t5
- Save address val + k * double offset into t4
- Load idx[k] into vector and change to 64 bits
- Load X[idx[k]] and A[k] into a vector.
- Call accumulate on loaded values A[k] and x[idx[k]]
- Bump k by the vector length
- Branch if our current k is less than ptr[i + 1] - 3

Vec_spmv_loop_tail:

- Does the same thing as the last loop but does one partial sum at a time

Vec_spmv_loop_end:

- Accumulates all partial sums and then stores into Y

<u>Optimizations Applied</u>

- Exterior loop unrolling: We unrolled the partial sums for one Y
- Interior loop unrolling: We unrolled the loop that does the partial sums
- Instruction order optimizations: We reordered the loads and the branching instructions in order to optimize for cycle count

<u>Performance/Benchmark Outputs</u>

- **<u>Unoptimized code</u>**

```
mcycle = 297341
minstret = 97714
BTB Accesses:             15681
BTB Hits:                 14493
BTB Hit Rate:             92.424%
BTB Mispredictions:       3462
BTB Misprediction Rate:   22.078%
BHT Accesses:             10614
BHT Mispredictions:       2211
BHT Misprediction Rate:   20.831%
RAS Accesses:             45
RAS Mispredictions:       0
RAS Misprediction Rate:   0.000%
L2$ Bytes Read:           243136
L2$ Bytes Written:        9344
L2$ Read Accesses:        3799
L2$ Write Accesses:       146
L2$ Read Misses:          2402
L2$ Write Misses:         0
L2$ Writebacks:           12
L2$ Miss Rate:            60.887%
D$ Bytes Read:            238748
D$ Bytes Written:         8955
D$ Read Accesses:         35413
D$ Write Accesses:        1236
D$ Read Misses:           3613
D$ Write Misses:          154
D$ Writebacks:            146
D$ Miss Rate:             10.279%
I$ Bytes Read:            347354
I$ Bytes Written:         0
I$ Read Accesses:         115010
I$ Write Accesses:        0
I$ Read Misses:           32
I$ Write Misses:          0
```

```
I$ Writebacks:           0
I$ Miss Rate:           0.028%
```

- **Optimized (vectorized/loop unroll) Code Output:**

```
mcycle = 176777
minstret = 31033
BTB Accesses:            11357
BTB Hits:              8265
BTB Hit Rate:           72.774%
BTB Mispredictions:        29
BTB Misprediction Rate:    0.255%
BHT Accesses:            6298
BHT Mispredictions:        74
BHT Misprediction Rate:    1.175%
RAS Accesses:            45
RAS Mispredictions:        0
RAS Misprediction Rate:    0.000%
L2$ Bytes Read:           239616
L2$ Bytes Written:        9088
L2$ Read Accesses:          3744
L2$ Write Accesses:        142
L2$ Read Misses:           2382
L2$ Write Misses:          0
L2$ Writebacks:           11
L2$ Miss Rate:           61.297%
D$ Bytes Read:           233560
D$ Bytes Written:        8947
D$ Read Accesses:          35764
D$ Write Accesses:        1228
D$ Read Misses:           3561
D$ Write Misses:          152
D$ Writebacks:           142
D$ Miss Rate:           10.037%
I$ Bytes Read:           146262
I$ Bytes Written:        0
I$ Read Accesses:          45010
I$ Write Accesses:        0
I$ Read Misses:           31
I$ Write Misses:          0
I$ Writebacks:           0
I$ Miss Rate:           0.069%
```

- **Optimized (reordered) Code Output:**

```
mcycle = 162269
minstret = 32032
BTB Accesses:           10857
BTB Hits:               7765
BTB Hit Rate:           71.521%
BTB Mispredictions:             29
BTB Misprediction Rate:         0.267%
BHT Accesses:           5799
BHT Mispredictions:             74
BHT Misprediction Rate:         1.276%
RAS Accesses:           45
RAS Mispredictions:             0
RAS Misprediction Rate:         0.000%
L2$ Bytes Read:         239808
L2$ Bytes Written:      9024
L2$ Read Accesses:              3747
L2$ Write Accesses:             141
L2$ Read Misses:                2382
L2$ Write Misses:       1
L2$ Writebacks:         7
L2$ Miss Rate:          61.291%
D$ Bytes Read:          229568
D$ Bytes Written:       8947
D$ Read Accesses:               35265
D$ Write Accesses:      1228
D$ Read Misses:         3564
D$ Write Misses:        152
D$ Writebacks:          141
D$ Miss Rate:           10.183%
I$ Bytes Read:          148260
I$ Bytes Written:       0
I$ Read Accesses:       45010
I$ Write Accesses:      0
I$ Read Misses:         31
I$ Write Misses:        0
I$ Writebacks:          0
I$ Miss Rate:           0.069%
```

- **Optimized (twice unrolled) Code Output:**

```
mcycle = 161387
minstret = 32032
BTB Accesses:           10861
BTB Hits:               7769
BTB Hit Rate:           71.531%
BTB Mispredictions:             29
BTB Misprediction Rate:         0.267%
BHT Accesses:           5803
BHT Mispredictions:             72
BHT Misprediction Rate:         1.241%
RAS Accesses:           45
RAS Mispredictions:             0
RAS Misprediction Rate:         0.000%
```

```
L2$ Bytes Read:        239424
L2$ Bytes Written:     9152
L2$ Read Accesses:             3741
L2$ Write Accesses:            143
L2$ Read Misses:               2384
L2$ Write Misses:      2
L2$ Writebacks:        6
L2$ Miss Rate:         61.432%
D$ Bytes Read:         229664
D$ Bytes Written:      8939
D$ Read Accesses:              35277
D$ Write Accesses:     1220
D$ Read Misses:        3554
D$ Write Misses:       153
D$ Writebacks:         143
D$ Miss Rate:          10.157%
I$ Bytes Read:         148228
I$ Bytes Written:      0
I$ Read Accesses:      45010
I$ Write Accesses:     0
I$ Read Misses:        34
I$ Write Misses:       0
I$ Writebacks:         0
I$ Miss Rate:          0.076%
```

<u>Performance Summary</u>

- From the base code to the vectorized loop unroll we have:
    - Dynamic Instruction improvement: 97714 to 31033
    - Cache Miss Rate improvement: 60.887% to 61.297%
    - Cycle improvement: 297341 to 176777
- From the vectorized loop unroll to the reordered we have:
    - Dynamic Instruction improvement: 31033 to 32032
    - Cache Miss Rate improvement: 61.297% to 61.291%
    - Cycle improvement: 176777 to 162269
- From the reordered to twice unrolled we have:
    - Dynamic Instruction improvement: 32032 to 32032
    - Cache Miss Rate improvement: 61.291% to 61.432%
    - Cycle improvement: 162269 to 161387

Code:

Optimized (vectorized/loop unroll) Code:

```
vec_spmv:
   add t6, zero, zero

vec_spmv_loop_i:
   lw t1, 0(a4) # Load current k into t1
   lw t2, 4(a4) # Load next k into t2
   #addi t2, t2, -3 # make the for loop limit ptr[i + 1] - 3
   sub t3, t2, t1
   vsetvli t0, t3, e64, m8
   fmv.d.x ft0, zero # Load zero into ft0
   vfmv.v.f v0, ft0 # make v0 the zero vector
   bge t1, t2, vec_tail

vec_spmv_loop_k:
   vsetvli zero, t3, e32, m4
   slli t4, t1, 3
   add t4, a1, t4
   slli t5, t1, 2
   add t5, a2, t5
   vle32.v v8, (t5)
   vwadd.vx v24, v8, x0 # idx[k]

   vsetvli zero, t3, e64, m8
   vsll.vi v24, v24, 3 # idx[k] << 3
   vle64.v v16, (t4) # load A
   vluxei64.v v24, (a3), v24 # load X
   vfmacc.vv v0, v16, v24 # accumulate into v0

   add t1, t1, t0 # make current k = k + 4
   blt t1, t2, vec_spmv_loop_k

 vec_tail:
   lw t2, 4(a4) # Load ptr[i + 1]
   bge t1, t2, vec_spmv_end

 vec_spmv_tail:
   slli t3, t1, 2
   add t4, a2, t3
   lw t4, 0(t4)
   slli t3, t1, 3 # t3 is K
   slli t4, t4, 3 # t4 is idx[k]

   add t3, a1, t3
   add t4, a3, t4
   fld ft3, (t3)
   fld ft4, (t4)
```

```
    fmul.d ft5, ft3, ft4
    fadd.d ft0, ft0, ft5

    addi t1, t1, 1
    blt t1, t2, vec_spmv_tail

vec_spmv_end:
    vfmv.s.f v20, ft0 # Load tail value into v20
    vfredosum.vs v0, v0, v20 # accumulate all partial sums into v0
    vfmv.f.s ft1, v0 # put sum into ft1
    fsd ft1, (a5) # store sum into y

    add a5, a5, 8 # bump y by double
    add a4, a4, 4 # bump ptr by int
    addi t6, t6, 1 # bump i by 1

    blt t6, a0, vec_spmv_loop_i

    ret
```

Optimized (reordered) Code:

```
vec_spmv:
        add t6, zero, zero

vec_spmv_loop_i:
        lw t1, 0(a4) # Load current k into t1
        lw t2, 4(a4) # Load next k into t2
        bge t1, t2, vec_tail
        sub t3, t2, t1
        vsetvli t0, t3, e64, m8
        fmv.d.x ft0, zero # Load zero into ft0
        vfmv.v.f v0, ft0 # make v0 the zero vector

vec_spmv_loop_k:
        slli t4, t1, 3
        add t4, a1, t4
        slli t5, t1, 2
        add t5, a2, t5
        vsetvli zero, t3, e64, m8
        vle64.v v16, (t4) # load A
        vsetvli zero, t3, e32, m4
        vle32.v v8, (t5)
        vwadd.vx v24, v8, x0 # idx[k]

        vsetvli zero, t3, e64, m8
        vsll.vi v24, v24, 3 # idx[k] << 3
        vluxei64.v v24, (a3), v24 # load X
        vfmacc.vv v0, v16, v24 # accumulate into v0
```

```
            add t1, t1, t0 # make current k = k + 4
            blt t1, t2, vec_spmv_loop_k

vec_tail:
            lw t2, 4(a4) # Load ptr[i + 1]
            bge t1, t2, vec_spmv_end

vec_spmv_tail:
            slli t3, t1, 2
            slli t3, t1, 3 # t3 is K
            add t3, a1, t3
            fld ft3, (t3)

            add t4, a2, t3
            lw t4, 0(t4)
            slli t4, t4, 3 # t4 is idx[k]

            add t4, a3, t4
            fld ft4, (t4)
            fmul.d ft5, ft3, ft4
            fadd.d ft0, ft0, ft5

            addi t1, t1, 1
            blt t1, t2, vec_spmv_tail

vec_spmv_end:
            add a4, a4, 4 # bump ptr by int
            addi t6, t6, 1 # bump i by 1
            vfmv.s.f v20, ft0 # Load tail value into v20
            vfredosum.vs v0, v0, v20 # accumulate all partial sums into v0
            vfmv.f.s ft1, v0 # put sum into ft1
            fsd ft1, (a5) # store sum into y

            add a5, a5, 8 # bump y by double

            blt t6, a0, vec_spmv_loop_i

            ret
```

Optimized (twice unrolled) Code:

```
vec_spmv:
            add t6, zero, zero

vec_spmv_loop_i:
            lw t1, 0(a4) # Load current k into t1
            lw t2, 4(a4) # Load next k into t2
            bge t1, t2, vec_spmv_end
```

```
        sub t3, t2, t1
        vsetvli t0, t3, e64, m8
        fmv.d.x ft0, zero # Load zero into ft0
        vfmv.v.f v0, ft0 # make v0 the zero vector

vec_spmv_loop_k:
        slli t4, t1, 3
        add t4, a1, t4
        slli t5, t1, 2
        add t5, a2, t5
        vsetvli zero, t3, e64, m8
        vle64.v v16, (t4) # load A
        vsetvli zero, t3, e32, m4
        vle32.v v8, (t5)
        vwadd.vx v24, v8, x0 # idx[k]

        vsetvli zero, t3, e64, m8
        #vle64.v v16, (t4) # load A
        vsll.vi v24, v24, 3 # idx[k] << 3
        vluxei64.v v24, (a3), v24 # load X
        vfmacc.vv v0, v16, v24 # accumulate into v0

        add t1, t1, t0 # make current k = k + 4
        blt t1, t2, vec_spmv_loop_k

vec_tail:
        lw t2, 4(a4) # Load ptr[i + 1]
        bge t1, t2, vec_spmv_end
        sub t3, t2, t1
        addi t3, t3, -1
        beqz t3, vec_tail_2
        addi t3, t3, 1

vec_spmv_loop_k_2:
        slli t4, t1, 3
        add t4, a1, t4
        slli t5, t1, 2
        add t5, a2, t5
        vsetvli zero, t3, e64, m4
        vle64.v v16, (t4) # load A
        vsetvli zero, t3, e32, m2
        vle32.v v8, (t5)
        vwadd.vx v24, v8, x0 # idx[k]

        vsetvli zero, t3, e64, m4
        vsll.vi v24, v24, 3 # idx[k] << 3
        vluxei64.v v24, (a3), v24 # load X
        vfmacc.vv v0, v16, v24 # accumulate into v0

        add t1, t1, t0 # make current k = k + 4
        blt t1, t2, vec_spmv_loop_k_2

vec_tail_2:
        bge t1, t2, vec_spmv_end

vec_spmv_tail:
        slli t3, t1, 2
        slli t3, t1, 3 # t3 is K
        add t3, a1, t3
        fld ft3, (t3)

        add t4, a2, t3
        lw t4, 0(t4)
        slli t4, t4, 3 # t4 is idx[k]

        add t4, a3, t4
        fld ft4, (t4)
        fmul.d ft5, ft3, ft4
        fadd.d ft0, ft0, ft5

        addi t1, t1, 1
        blt t1, t2, vec_spmv_tail

vec_spmv_end:
        add a4, a4, 4 # bump ptr by int
        addi t6, t6, 1 # bump i by 1
        vfmv.s.f v20, ft0 # Load tail value into v20
        vfredosum.vs v0, v0, v20 # accumulate all partial sums into v0
        vfmv.f.s ft1, v0 # put sum into ft1
        fsd ft1, (a5) # store sum into y

        add a5, a5, 8 # bump y by double

        blt t6, a0, vec_spmv_loop_i

        ret
```