# SIMD programming

# IAXPY (A|X| + |Y|) in…

## generalized CPU w/ modern ILP

| | | | |
|---|---|---|---|
| load x | load y | inc i | |
| mul | branch | load x | load y |
| add | mul | inc i | load x |
| store | add | branch | mul |
| store | load y | inc i | |
| add | branch | | |
| store | | | |

## SIMD units (also "array processors")

| | | |
|---|---|---|
| load x | load x | load x |
| mul | mul | mul |
| load y | load y | load y |
| add | add | add |
| store | store | store |

## Vector processors

| | | | | |
|---|---|---|---|---|
| load x | | | | |
| load x | mul | | | |
| load x | mul | load y | | |
| | mul | load y | add | |
| | | load y | add | store |
| | | | add | store |
| | | | | store |

Different implementation, same idea

**Benefits**:

- Easier dependence management
- No branches

# On larger data...

| load x | load x | load x | load x | load x | load x | load x | load x |
|---|---|---|---|---|---|---|---|
| mul | mul | mul | mul | mul | mul | mul | mul |
| load y | load y | load y | load y | load y | load y | load y | load y |
| add | add | add | add | add | add | add | add |
| store | store | store | store | store | store | store | store |

| | | | | |
|---|---|---|---|---|
| load x | | | | |
| load x | mul | | | |
| load x | mul | load y | | |
| load x | mul | load y | add | |
| load x | mul | load y | add | store |
| load x | mul | load y | add | store |
| load x | mul | load y | add | store |
| load x | mul | load y | add | store |
| | mul | load y | add | store |
| | | load y | add | store |
| | | | add | store |
| | | | | store |

# Terminology note

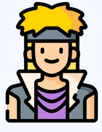
**SIMD** is an umbrella term describing architectures designed for DLP

Units that use parallelism to apply the same operation to different data

**Vectors** are an abstraction that defines data layout

SIMD architectures represent/understand these in different ways

**Marketing has made this a bit fuzzy**



Vector processors: classic supercomputers (heyday in the 80s) that operate on large vectors of data using SIMD principles

SIMD units: hardware in modern CPUs that computes on small vectors

Today: focusing on SIMD **operations** (hardware on Monday!)

? ? ?

How do we compile IAXPY for SIMD operations?

```
for (int i = 0; i < n; i++) {
   Y[i] = A * X[i] + Y[i];
}
```

# Non-vectorized IAXPY loop

$Y = \alpha X + Y \; (|X|, |Y| = 64)$

```
li s0, a # s0 = a
addi t0, s1, 256 # t0 = X + (64 * 4) (end address)
loop: lw t1, 0(s1) # t1 = x[i]
mul t1, t1, s0 # x[i] = x[i] * a
lw t2, 0(s2) # t2 = y[i]
add t2, t2, t1 # t2 = x[i] * a + y[i]
sw t2, 0(s2) # y[i] = t2 (x[i] * a + y[i])
addi s1, s1, 4 # increment x*
addi s2, s2, 4 # increment y*
bne s1, t0, loop
```

# Vector instructions (RISCV V ext)

Suffixes: .vv (vector-vector), .vx (vector-scalar), .vi (vector-immediate)

_Lots of operations_

At the minimum: load/store, operations on vectors

- Arithmetic/logical/shift: vadd, vsub, vrsub, etc
- Compare: vmseq, vmsne, vms{l,g}{t,e}[u]
- Max/min: vmin[u], vmax[u]
- Multiply-add (like dot product): vmacc, vnmsac, vmadd, vnmsub
- Reductions: vredsum, vredand, vredor, vredxor

# Vectorized IAXPY loop

```
li s0, a
addi t0, s1, 256
loop: lw t1, 0(s1)
mul t1, t1, s0
lw t2, 0(s2)
add t2, t2, t1
sw t2, 0(s2)
addi s1, s1, 4
addi s2, s2, 4
bne s1, t0, loop
```

```
li s0, a
vle32.v v0, s1 # v0 = X
vle32.v v1, s2 # v1 = Y
vmul.vx v0, v0, s0 # X = a * X
vadd.vv v1, v0, v1 # Y = a * X + Y
vse32.v v1, s2
```

OR JUST:

```
li s0, a
vle32.v v0, s1
vle32.v v1, s2
vmacc.vx v1, s0, v0 # Y = a * X + Y
vse32.v v1, s2
```

What assumptions are we making about our data here?

# How to handle a loop like this?

```c
for (int i = 0; i < 50; i++) {
    y[i] = a * x[i] + y[i];
}
```

```asm
vsetvli t0, s1, e32
# vl, t0 = min(MVL, s1)
li s0, a
vle32.v v0, s1
vle32.v v1, s2
vmacc.vx v1, s0, v0
vse32.v v1, s2
```

Used also in **strip mining**:

| Value of j | 0 | 1 | 2 | 3 | . . . | . . . | n/MVL |
|---|---|---|---|---|---|---|---|
| Range of i | 0 .. (m−1) | m .. (m−1) +MVL | (m+MVL) .. (m−1) +2×MVL | (m+2×MVL) .. (m−1) +3×MVL | . . . | . . . | (n−MVL) .. (n−1) |

vectorized inner loop

outer loop

**regular CPU 🤝 vector unit**

# What about these loops?

```
for (int i = 0; i < 64; i++) {
    if (x[i] != 0) {
        y[i] = a * x[i];
    }
}
```

```
for (int i = 0; i < n; i++) {
    X[m[i]] = X[m[i]] + Y[n[i]];
}
```

Masked/conditional instructions:

```
li s0, a
vle32.v v1, s1
vle32.v v2, s2
vmsne.vi v0, v1, 0
vmul.vx v2, v1, s0, v0.t
vse32.v v1, s2
```

**Gather**: collect all valid X[m[i]], Y[n[i]] in smaller vectors

**Scatter**: put the data back into X[m[i]], Y[n[i]]

In RVV: indexed load/stores, also the `vrgather` instruction

# Matrix multiplies

```
for (int i = 0; i < 128; i++) {
    for (int j = 0; j < 128; j++) {
        A[i][j] = 0;
        for (int k = 0; k < 128; k++) {
            A[i][j] += B[i][k] * C[k][j]
        }
    }
}
```

| B[0][0..127] | B[1][0..127] | B[2][0..127] | B[3][0..127] | ... |
|---|---|---|---|---|
| C[0][0..127] | C[1][0..127] | C[2][0..127] | C[3][0..127] | ... |

# Strided loads/stores

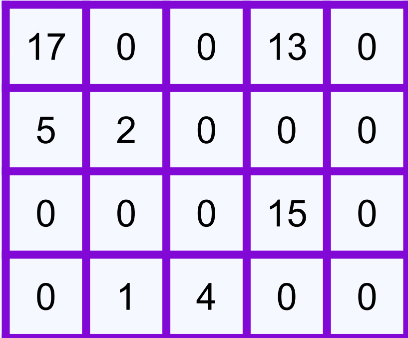
## 7.5. Vector Strided Instructions

```
# Vector strided loads and stores

# vd destination, rs1 base address, rs2 byte stride
vlse8.v    vd, (rs1), rs2, vm  #    8-bit strided load
vlse16.v   vd, (rs1), rs2, vm  #   16-bit strided load
vlse32.v   vd, (rs1), rs2, vm  #   32-bit strided load
vlse64.v   vd, (rs1), rs2, vm  #   64-bit strided load

# vs3 store data, rs1 base address, rs2 byte stride
vsse8.v    vs3, (rs1), rs2, vm  #    8-bit strided store
vsse16.v   vs3, (rs1), rs2, vm  #   16-bit strided store
vsse32.v   vs3, (rs1), rs2, vm  #   32-bit strided store
vsse64.v   vs3, (rs1), rs2, vm  #   64-bit strided store
```

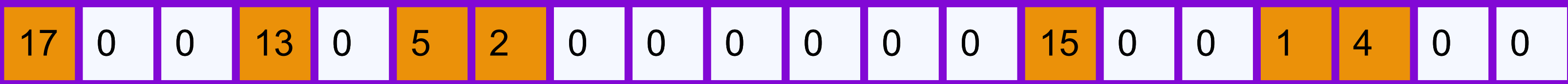
? ? ?

In what applications do sparse matrices appear in computer science?

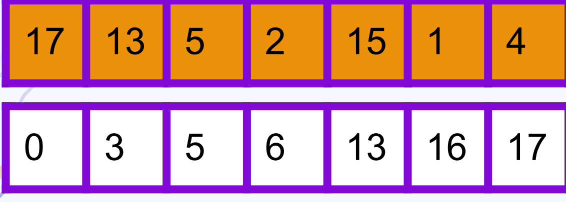
# Sparse data compression



| | | | | |
|---|---|---|---|---|
| 17 | 0 | 0 | 13 | 0 |
| 5 | 2 | 0 | 0 | 0 |
| 0 | 0 | 0 | 15 | 0 |
| 0 | 1 | 4 | 0 | 0 |



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | 0 | 0 | 13 | 0 | 5 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 1 | 4 | 0 | 0 |



| | | | | | | |
|---|---|---|---|---|---|---|
| 17 | 13 | 5 | 2 | 15 | 1 | 4 |
| 0 | 3 | 5 | 6 | 13 | 16 | 17 |

**"COO" (coordinate format)**

*can also be 2d*

Compressed formats don't just help us with space – they can also speed up some computations!

# Compact non-zero example

Page 82 of RVV spec (provided code also strip mines)

```
size_t compact_non_zero(
  size_t n,
  const int* in,
  int* out) {
    size_t i;
    size_t count = 0;
    int *p = out;
    for (i=0; i<n; i++) {
        const int v = *in++;
        if (v != 0)
            *p++ = v;
    }
    return (size_t) (p - out);
}
```

```
vle32.v v8, (a1)
vmsne.vi v0, v8, 0
vcpop.m a5, v0
viota.m v16, v0
vsll.vi v16, v16, 2, v0.t
vsuxei32.v v8, (a2), v16, v0.t
```

**a5 = 4**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **v8** | 17 | 0 | 0 | 13 | 0 | 5 | 2 | 0 |
| **v0** | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| **v16** | 0 | 1 | 1 | 4 | 2 | 8 | 12 | 4 |

(a2) (a2 + 4) (a2 + 8) (a2 + 12)

# Compiler effectiveness

| Processor | Compiler | Completely vectorized | Partially vectorized | Not vectorized |
|---|---|---|---|---|
| CDC CYBER 205 | VAST-2 V2.21 | 62 | 5 | 33 |
| Convex C-series | FC5.0 | 69 | 5 | 26 |
| Cray X-MP | CFT77 V3.0 | 69 | 3 | 28 |
| Cray X-MP | CFT V1.15 | 50 | 1 | 49 |
| Cray-2 | CFT2 V3.1a | 27 | 1 | 72 |
| ETA-10 | FTN 77 V1.0 | 62 | 7 | 31 |
| Hitachi S810/820 | FORT77/HAP V20-2B | 67 | 4 | 29 |
| IBM 3090/VF | VS FORTRAN V2.4 | 52 | 4 | 44 |
| NEC SX/2 | FORTRAN77 / SX V.040 | 66 | 5 | 29 |

**Figure G.9 Result of applying vectorizing compilers to the 100 FORTRAN test kernels.** For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

# Compiler🤝Programmer

| Benchmark name | Operations executed in vector mode, compiler-optimized | Operations executed in vector mode, with programmer aid | Speedup from hint optimization |
|---|---|---|---|
| BDNA | 96.1% | 97.2% | 1.52 |
| MG3D | 95.1% | 94.5% | 1.00 |
| FLO52 | 91.5% | 88.7% | N/A |
| ARC3D | 91.1% | 92.0% | 1.01 |
| SPEC77 | 90.3% | 90.4% | 1.07 |
| MDG | 87.7% | 94.2% | 1.49 |
| TRFD | 69.8% | 73.7% | 1.67 |
| DYFESM | 68.8% | 65.6% | N/A |
| ADM | 42.9% | 59.6% | 3.60 |
| OCEAN | 42.8% | 91.2% | 3.92 |
| TRACK | 14.4% | 54.6% | 2.52 |
| SPICE | 11.5% | 79.9% | 4.06 |
| QCD | 4.2% | 75.1% | 2.15 |

H&P Fig. 4.7

# More ergonomic solutions

Multimedia applications: people can use libraries

To get more flexibility than a library, ARM provides intrinsics (examples)