Vector Programming Model

Scalar Registers
- r15
- r0

Vector Registers
- v15
- v0
- [0] [1] [2] [VLRMAX-1]

Vector Length Register
- VLR

Vector Arithmetic Instructions
- ADDV v3, v1, v2

Vector Load and Store Instructions
- LV v1, r1, r2

Base, r1
Stride, r2
Memory

Vector Register
- v1
Vector Programming Model

Scalar Registers

Vector Registers

Vector Length Register

Vector Arithmetic Instructions

ADDV v3, v1, v2

Vector Load and Store Instructions

LV v1, r1, r2
# Vector Code Example

<table>
<thead>
<tr>
<th># C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;64; i++)</td>
</tr>
<tr>
<td>C[i] = A[i] + B[i];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Scalar Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LI R4, 64</td>
</tr>
<tr>
<td>loop:</td>
</tr>
<tr>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td>L.D F2, 0(R2)</td>
</tr>
<tr>
<td>ADD.D F4, F2, F0</td>
</tr>
<tr>
<td>S.D F4, 0(R3)</td>
</tr>
<tr>
<td>DADDIU R1, 8</td>
</tr>
<tr>
<td>DADDIU R2, 8</td>
</tr>
<tr>
<td>DADDIU R3, 8</td>
</tr>
<tr>
<td>DSUBIU R4, 1</td>
</tr>
<tr>
<td>BNEZ R4, loop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LI VLR, 64</td>
</tr>
<tr>
<td>LV V1, R1</td>
</tr>
<tr>
<td>LV V2, R2</td>
</tr>
<tr>
<td>ADDV.D V3, V1, V2</td>
</tr>
<tr>
<td>SV V3, R3</td>
</tr>
</tbody>
</table>
 Flynn’s Taxonomy

- Single instruction, single data (SISD)
  - E.g., our in-order processor

- Single instruction, multiple data (SIMD)
  - Multiple processing elements, same operation, different data
  - Vector
  - Multiple processing units execute the same instruction on different data in a lockstep. Either all complete or none do. Therefore, all units have to execute the same instruction at a given time

- Multiple instruction, multiple data (MIMD)
  - Multiple autonomous processors executing different instructions on different data
  - Most common and general parallel machine

- Multiple instruction, single data (MISD)
  - Why would anyone do this?
More Categories

- Single program, multiple data (SPMD)
  - Multiple autonomous processors execute the program at independent points
  - Difference with SIMD: SIMD imposes a lockstep
  - Programs at SPMD can be at independent points
  - SPMD can run on general purpose processors
  - Most common method for parallel computing

- Multiple program, multiple data (MPMD)
  - Multiple autonomous processors simultaneously operating at least 2 independent programs
Vector Instruction Set Advantages

- **Compact**
  - one short instruction encodes N operations

- **Expressive, tells hardware that these N operations:**
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory
    (unit-stride load/store)
  - access memory in a known pattern
    (strided load/store)

- **Scalable**
  - can run same code on more parallel pipelines (lanes)
T0 Vector Microprocessor (UCB/ICSI, 1995)

Vector register elements striped over lanes

Lane
Vector Startup

- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline). Some pipelines reduce control logic by requiring dead time between instructions to the same vector unit.
Dead Time and Short Vectors

Cray C90, Two lanes
4 cycle dead time
Maximum efficiency 94%
with 128 element vectors

T0, Eight lanes
No dead time
100% efficiency with 8 element vectors
Dead Time and Short Vectors

Cray C90, Two lanes
4 cycle dead time
Maximum efficiency 94%
with 128 element vectors

T0, Eight lanes
No dead time
100% efficiency with 8 element vectors
Vector Stripmining

**Problem:** Vector registers have finite length

**Solution:** Break loops into pieces that fit in registers, “Stripmining”

```assembly
ANDI R1, N, 63  # N mod 64
MTC1 VLR, R1    # Do remainder

for (i=0; i<N; i++)
    loop:
        LV V1, RA
        DSLL R2, R1, 3  # Multiply by 8
        DADDU RA, RA, R2 # Bump pointer
        LV V2, RB
        DADDU RB, RB, R2
        ADDV.D V3, V1, V2
        SV V3, RC
        DADDU RC, RC, R2
        DSUBU N, N, R1  # Subtract elements
        LI R1, 64
        MTC1 VLR, R1    # Reset full length
        BGTZ N, loop    # Any more to do?
```

A
B
C

for (i=0; i<N; i++)
    C[i] = A[i]+B[i];

Remainder

64 elements

Remainder

64 elements
Lab 4: Hwacha Vector-Fetch Machine

• Vector-Fetch Architecture
• More advanced/complicated vector programming model
  – Research into highly decoupled, efficient vector machines
  – Targeting mobile processors
    • Closest comparison is a phone GPU
• Hwacha.org
  – links to tech reports on ISA, microarch, etc.
Autovectorization Programming Model

SAXPY

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

- Mostly automatic, possibly some restructuring from the application writer
SAXPY mapped to SIMD Architecture

```plaintext
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases
```
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  v1w4 4t0, a2
  v1w4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases
SAXPY mapped to SIMD Architecture

\[
a_0: n, \ a_1: a, \ a_2: \ast x, \ a_3: \ast y
\]

stripmine:
- \( \text{vlw4 4t0, a2} \)
- \( \text{vlw4 4t1, a3} \)
- \( \text{vsplat4 4t2, a1} \)
- \( \text{vfma4 4t3, 4t2, 4t0, 4t1} \)
- \( \text{vsw4 4t3, a3} \)
- \( \text{add a2, a2, 4<<<2} \)
- \( \text{add a3, a3, 4<<<2} \)
- \( \text{sub a0, a0, 4} \)
- \( \text{bgte a0, 4, stripmine} \)

\ldots

handle edge cases
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: \*x, a3: \*y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine

handle edge cases
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases

SIMD
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases

SIMD
SAXPY mapped to SIMD Architecture

\[ a_0: n, a_1: a, a_2: *x, a_3: *y \]

stripmine:
\[
\begin{align*}
&\text{vlw4 4t0, a2} \\
&\text{vlw4 4t1, a3} \\
&\text{vsplat4 4t2, a1} \\
&\text{vfma4 4t3, 4t2, 4t0, 4t1} \\
&\text{vsw4 4t3, a3} \\
&\text{add a2, a2, 4<<2} \\
&\text{add a3, a3, 4<<2} \\
&\text{sub a0, a0, 4} \\
&\text{bgte a0, 4, stripmine} \\
\end{align*}
\]

\ldots \]

handle edge cases
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

```plaintext
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
handle edge cases
```

SIMD
SAXPY mapped to SIMD Architecture

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw8 8t0, a2
  vlw8 8t1, a3
  vsplat8 8t2, a1
  vfma8 8t3, 8t2, 8t0, 8t1
  vsw8 8t3, a3
  addi a2, a2, 8<<2
  addi a3, a3, 8<<2
  sub a0, a0, 8
  bgte a0, 8, stripmine
  . . .
  handle even more edge cases

SIMD

New and Improved SIMD
SIMD Arch. vs. Traditional Vector Arch.

SIMD

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...

handle edge cases

Traditional Vector

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
SIMD Arch. vs. Traditional Vector Arch.

SIMD

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases

Traditional Vector

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
a0: n, a1: a, a2: *x, a3: *y

**SIMD**

```assembly
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
.

handle edge cases
```

**Traditional Vector**

```assembly
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases

SIMD

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine

Traditional Vector
a0: n, a1: a, a2: *x, a3: *y

stripmine:
   vlw4 4t0, a2
   vlw4 4t1, a3
   vsplat4 4t2, a1
   vfma4 4t3, 4t2, 4t0, 4t1
   vsw4 4t3, a3
   add a2, a2, 4<<2
   add a3, a3, 4<<2
   sub a0, a0, 4
   bgte a0, 4, stripmine
   ...
   handle edge cases

SIMD

a0: n, a1: a, a2: *x, a3: *y

stripmine:
   vsetvl t0, a0
   vlw vv0, a2
   vlw vv1, a3
   vfma vv1, a1, vv0, vv1
   vsw vv1, a3
   slli t1, t0, 2
   add a2, a2, t1
   add a3, a3, t1
   sub a0, a0, t0
   bnez a0, stripmine

Traditional Vector
a0: n, a1: a, a2: *x, a3: *y

**stripmine:**
- \texttt{vlw4 4t0, a2}
- \texttt{vlw4 4t1, a3}
- \texttt{vsplat4 4t2, a1}
- \texttt{vfma4 4t3, 4t2, 4t0, 4t1}
- \texttt{vsw4 4t3, a3}
- \texttt{add a2, a2, 4<<2}
- \texttt{add a3, a3, 4<<2}
- \texttt{sub a0, a0, 4}
- \texttt{bgte a0, 4, stripmine}
  
  . . .
  
  handle edge cases

**SIMD**

\texttt{a0: n, a1: a, a2: *x, a3: *y}

**stripmine:**
- \texttt{vsetvl t0, a0}
- \texttt{vlw vv0, a2}
- \texttt{vlw vv1, a3}
- \texttt{vfma vv1, a1, vv0, vv1}
- \texttt{vsw vv1, a3}
- \texttt{slli t1, t0, 2}
- \texttt{add a2, a2, t1}
- \texttt{add a3, a3, t1}
- \texttt{sub a0, a0, t0}
- \texttt{bnez a0, stripmine}

**Traditional Vector**
SPMD Programming Model

- Classic model brought to forefront again by CUDA/OpenCL
- Same restructuring as autovectorization, more on top to get performance

```c
Kernel(int n, float a,
       float* x, float* y) {
    if (tid < n) {
        y[tid] = a*x[tid]+y[tid];
    }
}

Kernel<<<n/32*32>>>(
    n, a, x, y);
```
SPMD Programming Model

- Classic model brought to forefront again by CUDA/OpenCL
- Same restructuring as auto-vectorization, more on top to get performance

```
Kernel(int n, float a,
       float* x, float* y) {
    if (tid < n) {
        y[tid] = a*x[tid]+y[tid];
    }
}

Kernel<<<n/32*32>>>(n, a, x, y);
```
SPMD Programming Model

- Classic model brought to forefront again by CUDA/OpenCL
- Same restructuring as autovectorization, more on top to get performance

```c
Kernel(int n, float a, 
      float* x, float* y) {
    if (tid < n) {
        y[tid] = a*x[tid]+y[tid];
    }
}

Kernel<<<n/32*32>>> 
(n, a, x, y);
```
a0: n, a1: a,  
a2: *x, a3: *y

mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)
skip:
stop
a0: n, a1: a,  
a2: *x, a3: *y
  
  mv t0, tid  
bge t0, n, skip  
slli t0, t0, 2  
add a2, a2, t0  
add a3, a3, t0  
lw t1, 0(a2)  
lw t2, 0(a3)  
fma.s t0, a1, t1, t2  
sw t0, 0(a3)  
  
skip:  
  stop
SIMT Architecture

\[ a_0: n, a_1: a, \]
\[ a_2: x, a_3: y \]

\[ \text{mv } t0, \text{tid} \]
\[ \text{bge } t0, n, \text{skip} \]
\[ \text{slli } t0, t0, 2 \]
\[ \text{add } a_2, a_2, t0 \]
\[ \text{add } a_3, a_3, t0 \]
\[ \text{lw } t1, 0(a_2) \]
\[ \text{lw } t2, 0(a_3) \]
\[ \text{fma.s } t0, a_1, t1, t2 \]
\[ \text{sw } t0, 0(a_3) \]

\text{skip:}
\text{stop}

\text{SIMT}
a0: n, a1: a,
a2: *x, a3: *y

mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)
skip:
    stop
a0: n, a1: a,
a2: *x, a3: *y

mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)
skip:
  stop
SIMT Architecture

a0: n, a1: a,
a2: *x, a3: *y

mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)

skip:
stop
SIMT Arch. vs. Traditional Vector Arch.

SIMT

\[ a_0: n, a_1: a, a_2: *x, a_3: *y \]

\[
\begin{align*}
&\text{mv } t_0, \text{tid} \\
&\text{bge } t_0, n, \text{skip} \\
&\text{slli } t_0, t_0, 2 \\
&\text{add } a_2, a_2, t_0 \\
&\text{add } a_3, a_3, t_0 \\
&\text{lw } t_1, 0(a_2) \\
&\text{lw } t_2, 0(a_3) \\
&\text{fma.s } t_0, a_0, t_1, t_2 \\
&\text{sw } t_0, 0(a_3) \\
&\text{skip:} \\
&\text{stop}
\end{align*}
\]

Traditional Vectors

\[ a_0: n, a_1: a, a_2: *x, a_3: *y \]

\[
\begin{align*}
&\text{stripmine:} \\
&\quad \text{vsetvl } t_0, a_0 \\
&\quad \text{vlw } vr_0, a_2 \\
&\quad \text{vlw } vr_1, a_3 \\
&\quad \text{vfma } vr_1, a_1, vr_0, vr_1 \\
&\quad \text{vsw } vr_1, a_3 \\
&\quad \text{slli } t_1, t_0, 2 \\
&\quad \text{add } a_2, a_2, t_1 \\
&\quad \text{add } a_3, a_3, t_1 \\
&\quad \text{sub } a_0, a_0, t_0 \\
&\quad \text{bnez } a_0, \text{stripmine}
\end{align*}
\]
a0: n, a1: a, a2: *x, a3: *y

SIMT

```assembly
mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a0, t1, t2
sw t0, 0(a3)
```

```
skip:
  stop
```

Traditional Vectors

```assembly
stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```
SIMT Arch. vs. Traditional Vector Arch.

**SIMT**

```
a0: n, a1: a, a2: *x, a3: *y

mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a0, t1, t2
sw t0, 0(a3)

skip:
    stop
```

**Traditional Vectors**

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
    vsetvl t0, a0
    vlw vr0, a2
    vlw vr1, a3
    vfma vr1, a1, vr0, vr1
    vsw vr1, a3
    slli t1, t0, 2
    add a2, a2, t1
    add a3, a3, t1
    sub a0, a0, t0
    bnez a0, stripmine
```
SIMT Arch. vs. Traditional Vector Arch.

**SIMT**

\[
a0: n, \ a1: a, \ a2: \,*x, \ a3: \,*y
\]

```
mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a0, t1, t2
sw t0, 0(a3)
```

`skip:`

```
stop
```

**Traditional Vectors**

\[
a0: n, \ a1: a, \ a2: \,*x, \ a3: \,*y
\]

```
stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```
SIMT Arch. vs. Traditional Vector Arch.

**SIMT**

```assembly
a0: n, a1: a, a2: *x, a3: *y

mv t0, tid
bge t0, n, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a0, t1, t2
sw t0, 0(a3)
```

```
skip:
    stop
```

**Traditional Vectors**

```assembly
a0: n, a1: a, a2: *x, a3: *y

stripmine:
    vsetvl t0, a0
    vlw vr0, a2
    vlw vr1, a3
    vfma vr1, a1, vr0, vr1
    vsw vr1, a3
    slli t1, t0, 2
    add a2, a2, t1
    add a3, a3, t1
    sub a0, a0, t0
    bnez a0, stripmine
```

But how could we do vectors better?
Hwacha Summary

- Push in-order vector design to efficiency limit
- Designed as a non-standard RISC-V extension to hang off Rocket
- Shared memory space with Rocket
- Designed to work with the OS with restartable exceptions
- And much more...
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

```
a0: n, a1: a,
a2: *x, a3: *y

vmss vs0, a1
stripmine:
vsetvl t0, a0
vmsa va0, a2
vmsa va1, a3
vf saxpy
slli t1, t0, 2
add a2, a2, t1
add a3, a3, t1
sub a0, a0, t0
bnez a0, stripmine
```

Worker Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, val
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, val
  vstop
```
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

```
a0: n, a1: a,
a2: *x, a3: *y

vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Worker Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

\[
a0: n, \ a1: a, \\
a2: \ *x, \ a3: \ *y
\]

\[
\begin{align*}
\text{vmss} & \ vs0, \ a1 \\
\text{stripmine:} & \\
\text{vsetvl} & \ t0, \ a0 \\
\text{vmsa} & \ va0, \ a2 \\
\text{vmsa} & \ va1, \ a3 \\
\text{vf} & \ \text{saxpy} \\
\text{slli} & \ t1, \ t0, \ 2 \\
\text{add} & \ a2, \ a2, \ t1 \\
\text{add} & \ a3, \ a3, \ t1 \\
\text{sub} & \ a0, \ a0, \ t0 \\
\text{bnez} & \ a0, \ \text{stripmine}
\end{align*}
\]

Worker Thread

\[
saxpy:
\begin{align*}
\text{vlw} & \ vv0, \ va0 \\
\text{vlw} & \ vv1, \ va1 \\
\text{vfma.svv} & \ vv1, \ vs0, \ vv0, \ vv1 \\
\text{vsw} & \ vv1, \ va1 \\
\text{vstop}
\end{align*}
\]
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

\[
\begin{align*}
a0 &: n, a1 &: a, \\
a2 &: *x, a3 &: *y
\end{align*}
\]

\begin{itemize}
  \item vmss vs0, a1
  \item stripmine:
    \begin{itemize}
    \item vsetvl t0, a0
    \item vmsa va0, a2
    \item vmsa val, a3
    \item vf saxpy
    \item slli t1, t0, 2
    \item add a2, a2, t1
    \item add a3, a3, t1
    \item sub a0, a0, t0
    \item bnez a0, stripmine
    \end{itemize}
\end{itemize}

Worker Thread

\begin{itemize}
  \item saxpy:
    \begin{itemize}
    \item vlw vv0, va0
    \item vlw vv1, val
    \item vfma.svv vv1, vs0, vv0, vv1
    \item vsw vv1, val
    \item vstop
    \end{itemize}
\end{itemize}
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

```
a0: n, a1: a,
a2: *x, a3: *y

  vmss vs0, a1
  stripmine:
    vsetvl t0, a0
    vmsa va0, a2
    vmsa va1, a3
    vf saxpy
    slli t1, t0, 2
    add a2, a2, t1
    add a3, a3, t1
    sub a0, a0, t0
    bnez a0, stripmine
```
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

a0: n, a1: a,
a2: *x, a3: *y

vmss vs0, a1
stripmine:
  vsetvl t0, a0
vmsa va0, a2
vmsa va1, a3
vf saxpy
slli t1, t0, 2
add a2, a2, t1
add a3, a3, t1
sub a0, a0, t0
bnez a0, stripmine

saxpy:
  vlw vv0, va0
  vlw vv1, val
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, val
  vstop

Worker Thread

CP

Vector Execution Unit

Vector Unit
Hwacha Vector-Fetch Architectural Paradigm

Control Thread

\[\begin{align*}
a0: n, & \quad a1: a, \\
a2: *x, & \quad a3: *y
\end{align*}\]

- \text{vmss vs0, a1}
- \text{stripmine:}
  - \text{vsetvl t0, a0}
  - \text{vmsa va0, a2}
  - \text{vmsa va1, a3}
  - \text{vf saxpy}
  - \text{slli t1, t0, 2}
  - \text{add a2, a2, t1}
  - \text{add a3, a3, t1}
  - \text{sub a0, a0, t0}
  - \text{bnez a0, stripmine}

saxpy:
- \text{vlw vv0, va0}
- \text{vlw vv1, va1}
- \text{vfma.svv vv1, vs0, vv0, vv1}
- \text{vsw vv1, va1}
- \text{vstop}

Worker Thread

Diagram of CP, Vector Execution Unit, Vector Unit, and VI$
Hwacha Vector-Fetch Architectural Paradigm

**Control Thread**

\[ a0: n, a1: a, \]
\[ a2: *x, a3: *y \]

\[ \text{vmss vs0, a1} \]
\[ \text{stripmine:} \]
\[ \text{vsetvl t0, a0} \]
\[ \text{vmsa va0, a2} \]
\[ \text{vmsa va1, a3} \]
\[ \text{vf saxpy} \]
\[ \text{slli t1, t0, 2} \]
\[ \text{add a2, a2, t1} \]
\[ \text{add a3, a3, t1} \]
\[ \text{sub a0, a0, t0} \]
\[ \text{bnez a0, stripmine} \]

**Worker Thread**

\[ \text{saxpy:} \]
\[ \text{vlw vv0, va0} \]
\[ \text{vlw vv1, val} \]
\[ \text{vfma.svv vv1, vs0, vv0, vv1} \]
\[ \text{vsw vv1, val} \]
\[ \text{vstop} \]
Hwacha Compilers

- **SPMD-style programming model**
  - Update our OpenCL compiler in LLVM to new ISA
  - With predication pass and scalarization pass

- **Autovectorization programming model**
  - Support exists in LLVM infrastructure
  - Previous autovectorization work from ParLab

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

Kernel(int n, float a,
       float* x, float* y) {
    if (tid < n) {
        y[tid] = a*x[tid]+y[tid];
    }
}

Kernel<<<n/32*32>>> (n, a, x, y);
```
Decoupling – SAXPY Example

Control Processor (CP)

Vector Prefetch Unit (VPU)

Vector Execution Unit (VXU)

Vector Memory Unit (VMU)

outer memory system (L2$)

A SPIRE
Decoupling – SAXPY Example

- Before stripmine loop, control thread pushes scalar register write to VXU command queue
Decoupling – SAXPY Example

- Vector length adjustments, address register writes, and vector fetch commands pushed to both queues for each stripmine iteration
Decoupling – SAXPY Example

- VPU prefetched instruction and pre-executes vector loads, refilling cache ahead of VXU access
**Decoupling – SAXPY Example**

- VMU performs regular memory operations when data is eventually needed; overlaps with VXU computation.
Decoupling – SAXPY Example

- Control processor runs ahead of vector unit to next iteration
  - Performs address computations
  - Pushes next vector fetch command
Decoupling – SAXPY Example

- VPU prefetches next iteration while VXU remains busy with previous one
Vector Lane Organization

- Compact register file of four 1R1W SRAM banks
- Per-bank integer ALU
- Two independently scheduled FMA clusters
  - Total of four double-precision FMAs per cycle
- Pipelined integer multiplier
- Variable-latency decoupled functional units
  - Integer divide
  - Floating-point divide with square root
Systolic Bank Execution

- Sustains \( n \) operands/cycle after \( n \)-cycle initial latency

- “Fire and forget” after hazards are cleared upon sequencing
- Chaining follows naturally from interleaving \( \mu \)ops belonging to dependent instructions
Reconfigurable Vector Register File

- Programming model allows specifying number of architectural registers
- Maximum hardware vector length automatically extends to fill the capacity of the register file

|-------|-------|-------|-------|-------|-------|-------|-------|

- Exchange unused architectural registers for longer hardware vectors

vsetcfg 4
vlen = 2

vsetcfg 2
vlen = 4
Mixed-Precision Support

- Hardware can subdivide a physical register into multiple narrower architectural registers as needed
  - Subword packing transparent to software
  - Improved utilization of operand communication bandwidth
  - Spatial functional unit parallelism

```
vv0[0]  vv0[0]
vv0[1]  vv0[1]
vv1[0]  vv0[2]
vv1[1]  vv0[3]
vv2[0]  vv1[0]
vv2[1]  vv1[1]
vv3[0]  vv1[2]
vv3[1]  vv1[3]
```

vsetcfg 1, 1
vlen = 5