Computer Architecture

SIMD Architecture: Vector Processing

Debadatta Mishra, CSE, IITK

Classification of Computing Frameworks

- Single Instruction Single Data (SISD)
  - Single Core with ILP techniques such as superscalar and speculative execution

Covered

- Single Instruction Multiple Data (SIMD)
  - Multiple SIMD processor execute the same instruction on multiple data to e
    data level parallelism

Agenda for remaining lectures

- Multiple Instruction Single Data (MISD)
  - Not very common as concurrent operation on the same data ensuring correctness is difficult to achieve

- Multiple Instruction Multiple Data (MIMD)
  - Independent processing by different processors with correctness guarantees
  - Multi-core and multi-threaded processors

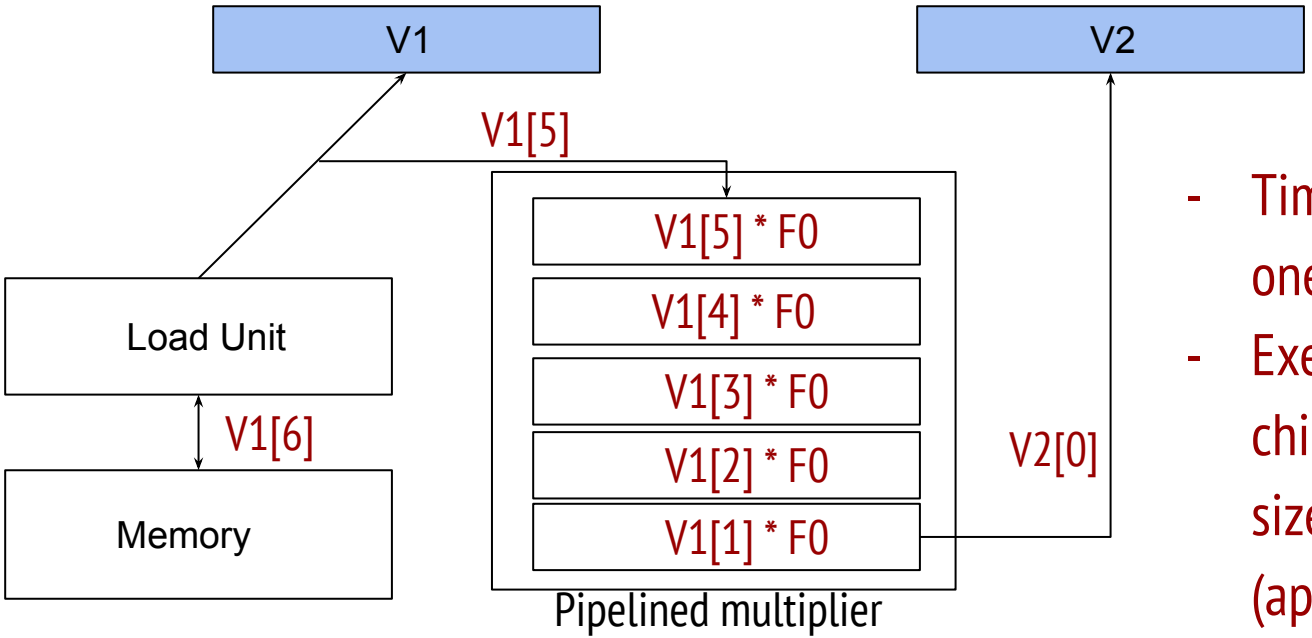
CS423

Vector processing

```
ldV v1,r1
mulVS v2,v1,f0
```

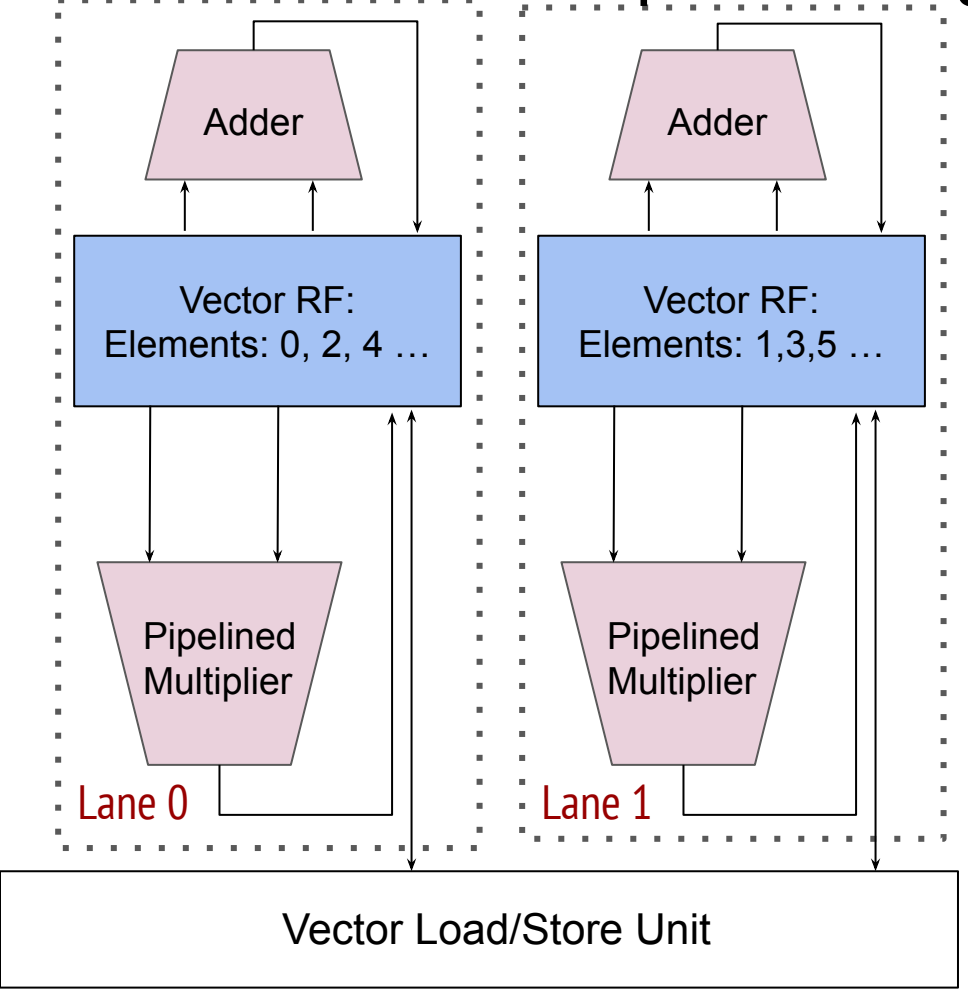
```
//load X to vector reg v1
//v2 = a*X
```

- Assuming single lane, one element in V2 is ready in each cycle



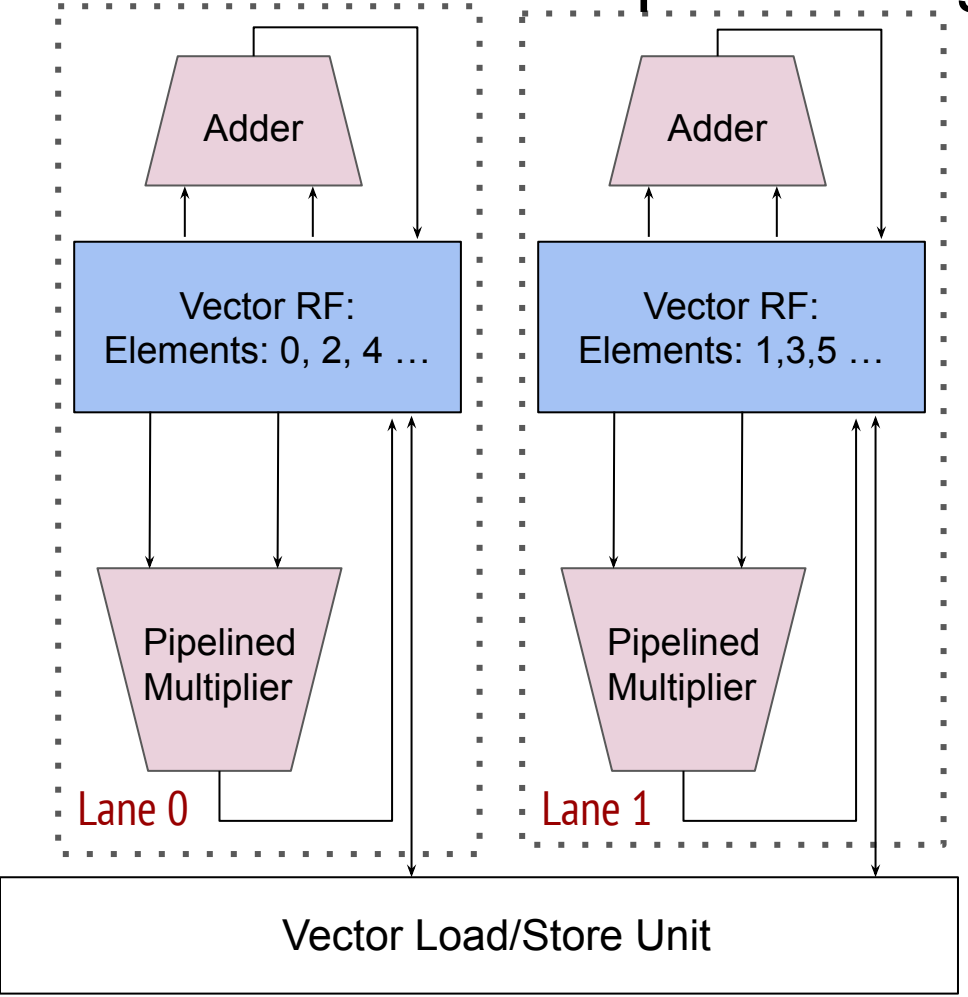
- Time taken to execute one convoy = chime
- Execution time of a chime for a vector of size 64 = 64 cycles (approx.)

Multi-lane vector processing



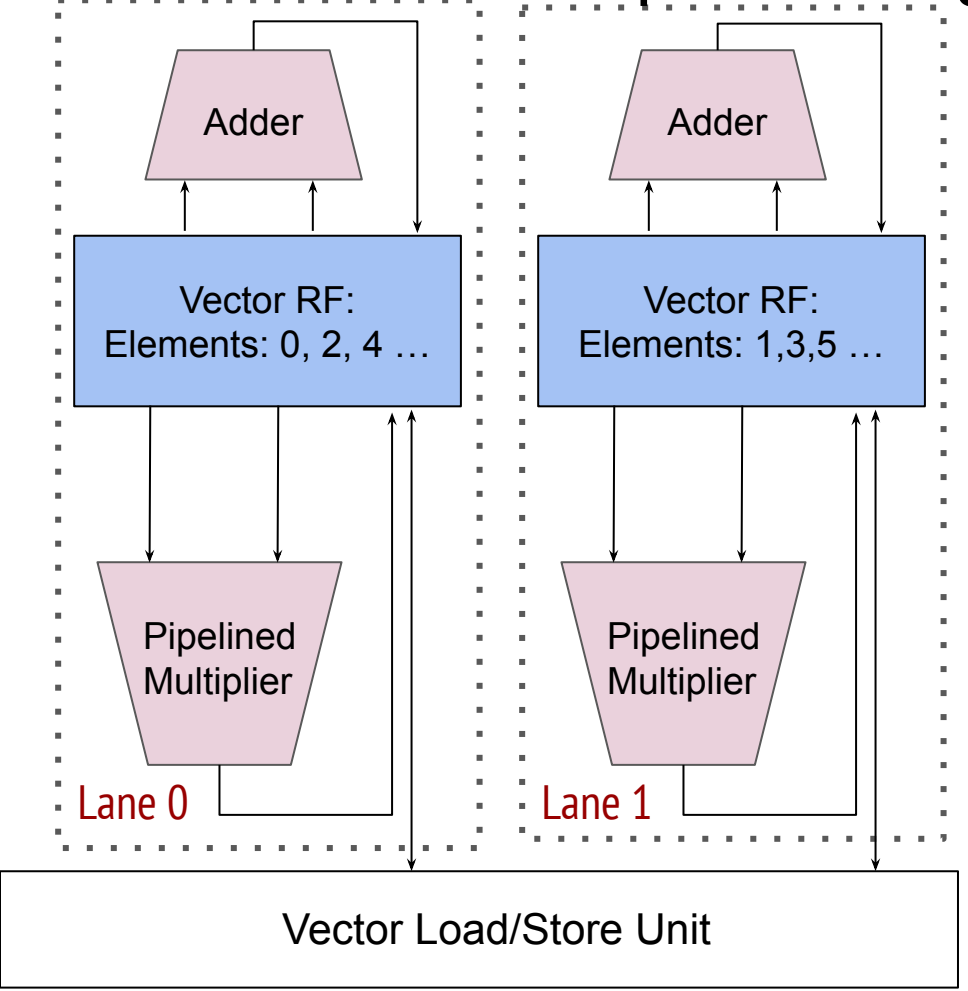
- Each lane consists of one execution pipeline and a portion of vector RF
- Example: Lane 0 consists of the first pipeline of all FUs and first element of all vector registers
- Benefits?
- Is there any interdependence?
- How about #of read/write ports?

Multi-lane vector processing



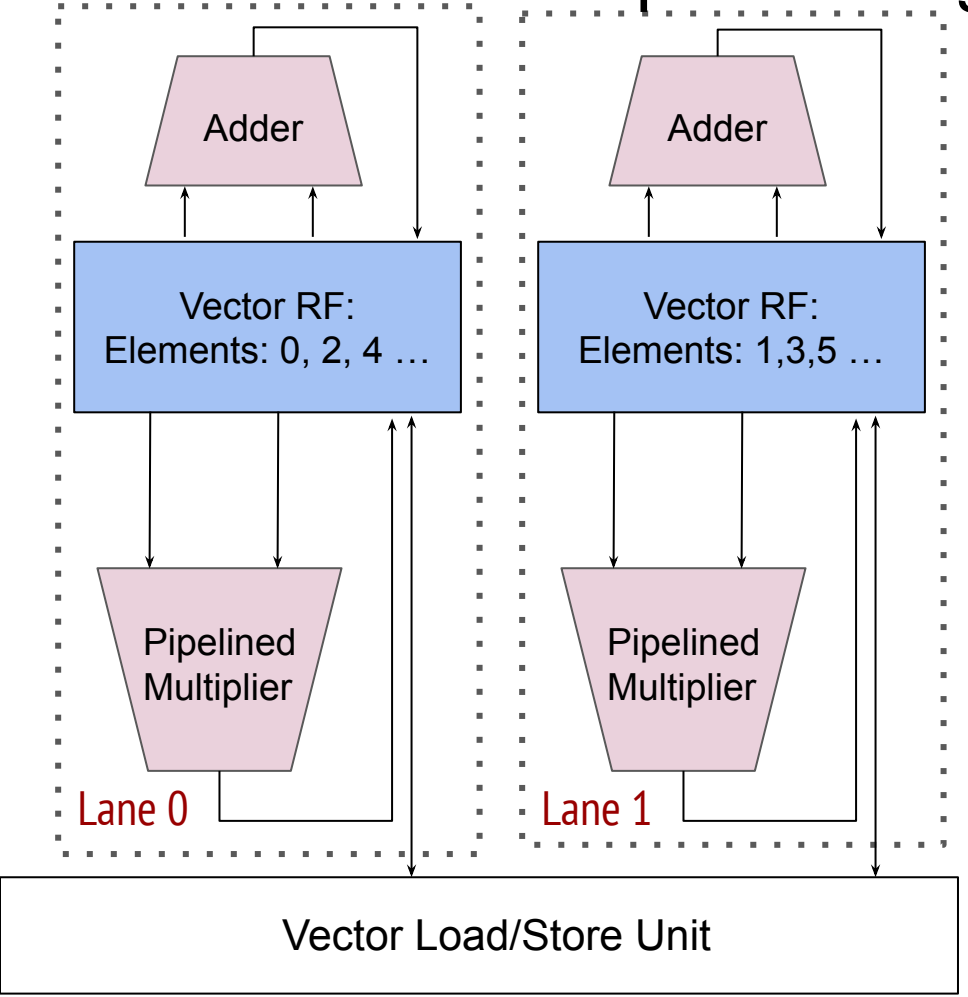
- Each lane consists of one execution pipeline and a portion of vector RF
- Example: Lane 0 consists of the first pipeline of all FUs and first element of all vector registers
- Benefits? Chime execution time reduced by a factor of #of lanes
- Is there any interdependence?
- How about #of read/write ports?

Multi-lane vector processing



- Each lane consists of one execution pipeline and a portion of vector RF
- Example: Lane 0 consists of the first pipeline of all FUs and first element of all vector registers
- Benefits? Chime execution time reduced by a factor of #of lanes
- Is there any interdependence? Lanes can execute independently
- How about #of read/write ports?

Multi-lane vector processing



- Each lane consists of one execution pipeline and a portion of vector RF
- Example: Lane 0 consists of the first pipeline of all FUs and first element of all vector registers
- Benefits? Chime execution time reduced by a factor of #of lanes
- Is there any interdependence? Lanes can execute independently
- How about #of read/write ports? No increase due to inter-lane interaction

Handling arbitrary length vectors

- Vector length can be more than maximum # of data elements in a VR (a.k.a. MVL)
- Vector length may not be a multiple of MVL
- Strip mining
  - Split the vector into multiple of MVL
  - Residual elements handled separately
- Example: Multiply two vectors of length 1000
  - 15 iterations with full sized vectors (VLEN = 64)
  - One iteration with VLEN = 40

Conditional vector operations using mask registers

Conditional Execution

```
for(i=0; i<64; i=i+1)
     if(x[i] != 0)
        y[i] = y[i] / x[i]
```

- The loop can not be vectorized
- Compiler can use a vector mask register (VMR) along with special instructions for conditional execution

Conditional vector operations using mask registers

Conditional Execution

```
for(i=0; i<64; i=i+1)
     if(x[i] != 0)
        y[i] = y[i] / x[i]
```

```
// r1 = &x[0], r2 = &y[0], f0 = 0 (FP)
ldV v1, r1 //load x into v1
ldV v2, r2 //load y into v2
sneVS v1, f0 //set vmr[i] = 1 if v1[i] != 0
divVV v2,v2,v1 // v2=v2/v1 under vmr
sV v2, r2 // store v to y
```

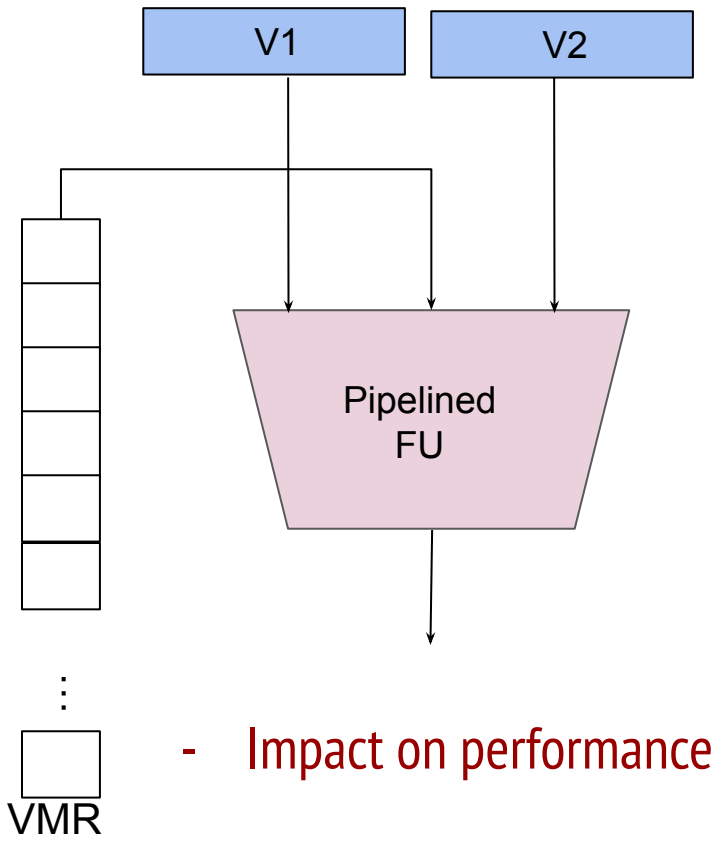
- The loop can not be vectorized
- Compiler can use a vector mask register (VMR) along with special instructions for conditional execution
- In vector processors VMR is part of architectural state – compiler should set it appropriately

Masked vector operation

Conditional Execution

```
for(i=0; i<64; i=i+1)
     if(x[i] != 0)
        y[i] = y[i] / x[i]
```

```
// r1 = &x[0], r2 = &y[0], f0 = 0 (FP)
ldV v1, r1 //load x into v1
ldV v2, r2 //load y into v2
sneVS v1, f0 //set vmr[i] = 1 if v1[i] != 0
divVV v2,v2,v1 // v2=v2/v1 under vmr
sV v2, r2 // store v to y
```



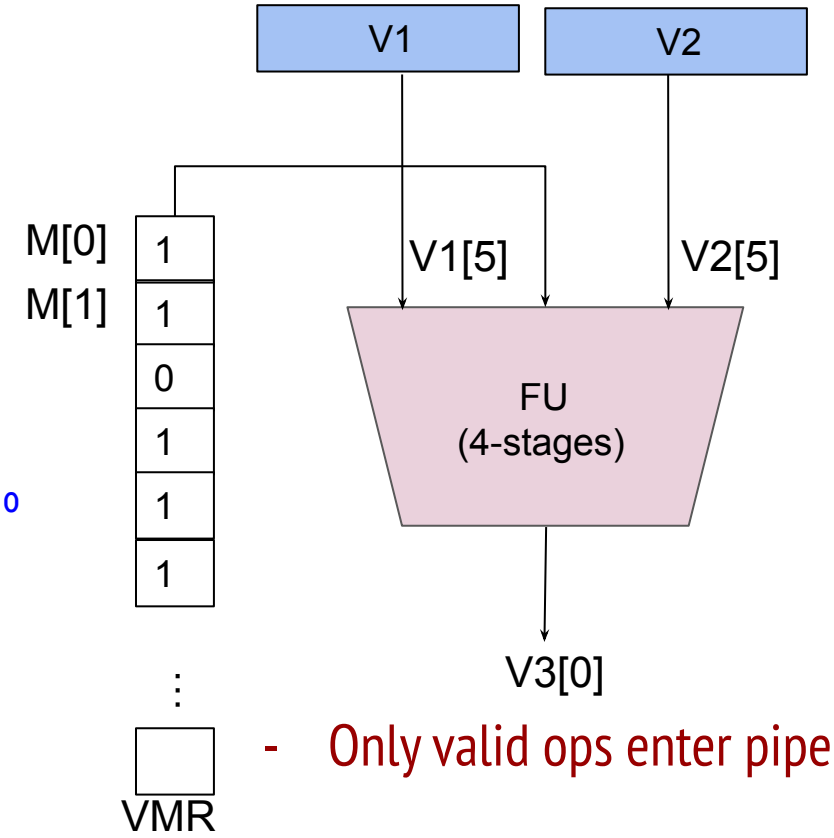
- Impact on performance?

Masked vector operation

Conditional Execution

```
for(i=0; i<64; i=i+1)
      if(x[i] != 0)
         y[i] = y[i] / x[i]
```

```
// r1 = &x[0], r2 = &y[0], f0 = 0 (FP)
ldV v1, r1 //load x into v1
ldV v2, r2 //load y into v2
sneVS v1, f0 //set vmr[i] = 1 if v1[i] != 0
divVV v2,v2,v1 // v2=v2/v1 under vmr
sV v2, r2 // store v to y
```



- Only valid ops enter pipeline

Sparse data processing

Pack using indirection

```
for(i=0; i<64; i=i+1)
        A[K[i]] = A[K[i]] + B[M[i]]
```

- K and M contain index values for non-zero elements of A and B respectively
- How vector operations can be performed?

Sparse data processing

Pack using indirection

```
for(i=0; i<64; i=i+1)
        A[K[i]] = A[K[i]] + B[M[i]]
```

- K and M contain index values for non-zero elements of A and B respectively
- How vector operations can be performed?
  - Gather: Uses an index vector to load non-zero elements
  - Perform vector computation
  - Scatter: Use index vector to store non-zero elements

Sparse data processing

Pack using indirection

```
for(i=0; i<64; i=i+1)
        A[K[i]] = A[K[i]] + B[M[i]]
```

```
//Assuming same sparseness

ldV Vk, Rk    //load index vector to Vk register
ldVI Va, (Ra+Vk)    // Gather A and pack into Va
ldV Vm, Rm    //load index vector to Vm register
ldVI Vb, (Rb+Vm)   // Gather B and pack into Vb
addVV Va,Va,Vb     // v2=v2 + v1 under vmr
sV (Ra+Vk), Va     // store v to y
```

- K and M contain index values for non-zero elements of A and B respectively
- Performance of gather-scatter vs. non-indexed implementation?

Sparse data processing

Pack using indirection

```
for(i=0; i<64; i=i+1)
        A[K[i]] = A[K[i]] + B[M[i]]
```

```
//Assuming same sparseness

ldV Vk, Rk    //load index vector to Vk register
ldVI Va, (Ra+Vk)    // Gather A and pack into Va
ldV Vm, Rm    //load index vector to Vm register
ldVI Vb, (Rb+Vm)   // Gather B and pack into Vb
addVV Va,Va,Vb     // v2=v2 + v1 under vmr
sV (Ra+Vk), Va     // store v to y
```

- K and M contain index values for non-zero elements of A and B respectively
- Performance of gather-scatter vs. non-indexed implementation?
  - Gather-scatter avoids unnecessary operations
  - Specialized memory access mechanisms required to serve indexed access efficiently

SIMD extensions for Multimedia

- Motivation: Multimedia systems require fewer bits (8 or 16 bits) to represent color, transparency, audio samples etc.
- Partitioning support in functional units can allow simultaneous operations
  - Example: A partitioned 256 bit adder can add vectors of 32 8-bit operands
- Few additional registers may be provided to hold the vectors

Example with 256-bit SIMD extension

```
// Y = a * x + Y
// r1 = &x[0], r2 = &y[0],

ldD f0,(a)           //load scalar a
mov f1,f0
mov f2,f0            // copy to allow SIMD op
mov f3,f0
daddiu r3,r1, #512    //r3=&X[0] + 512
loop: ldDE f4,0(r1)   //f[4-7] = x[i..i+3]
mulDE f4,f4,f0        //f[4-7] = a*x[i..i+3]
ldDE f8,0(r2)         //f[8-11] = y[i..i+3]
addDE f8,f8,f4        //f[8-11] = a*x[]+y[]
sDE f8,0(r2)          //y[i..i+3] = f[8-11]
daddiu r1,r1,#32      // r1 = &x[i+4]
daddiu r2,r2,#32      // r2 = &y[i+4]
sub r4,r3,r1          // elements remaining
benz r4, loop
```

- Semantic

  “mulDE f4, f4,f0” performs vector multiplication between registers f[0-3] and f[4-7] and store the results in f[4-7]

- # of iterations reduced by a factor of 4

SIMD extensions for Multimedia

- Motivation: Graphics systems require fewer bits (8 or 16 bits) to represent color, transparency etc.
- Partitioning support in functional units can allow simultaneous operations
  - Example: A partitioned 256 bit adder can add vectors of 32 8-bit operands
- Few additional registers may be provided to hold the vectors
- Examples: Intel MMX →SSE* →AVX to support specialized library development
- Limitations
  - Limited support for addressing modes such as gather-scatter
  - Typically does not provide mask registers for conditional execution

SIMD extensions for Multimedia

- Motivation: Graphics systems require fewer bits (8 or 16 bits) to represent color, transparency etc.
- Partitioning support in functional units can allow simultaneous operations
  - Example: A partitioned 256 bit adder can add vectors of 32 8-bit operands
- Few additional registers may be provided to hold the vectors
- Examples: Intel MMX →SSE* →AVX to support specialized library development
- Limitations
  - Limited support for addressing modes such as gather-scatter
  - Typically does not provide mask registers for conditional execution
- Advantages?

SIMD extensions for Multimedia

- Motivation: Graphics systems require fewer bits (8 or 16 bits) to represent color, transparency etc.
- Partitioning support in functional units can allow simultaneous operations
  - Example: A partitioned 256 bit adder can add vectors of 32 8-bit operands
- Few additional registers may be provided to hold the vectors
- Examples: Intel MMX $\rightarrow$SSE* $\rightarrow$AVX to support specialized library development
- Limitations
  - Limited support for addressing modes such as gather-scatter
  - Typically does not provide mask registers for conditional execution
- Advantages? Resource reuse, fast context switches, little additional cost for protection checks, suitable for small vector operations