Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures.
We will cover sections 4.1, 4.2, 4.3, and 4.5 and delay the coverage of GPUs (section 4.5)

Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- SIMD allows programmer to continue to think sequentially
**SIMD Parallelism**

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

**Vector Architectures**

- Basic idea:
  - Read sets of data elements into "vector registers"
  - Operate on those registers
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth
RV64V

- Example architecture: RV64V
  - Loosely based on Cray-1
  - Vector registers
    - Each register holds a vector (32 elements, each 64 bits long)
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers

RV64V Instructions

- vadd.vv: add two vectors
- vadd.sv: add a scaler to a vector
- vld, vst: vector load and vector store from address

Example: DAXPY

```
fld F0,a ; load scalar a
vld V1,Rx ; load vector X
vmul.vs V2,V1,F0 ; vector-scalar multiply
vld V3,Ry ; load vector Y
vadd.vv V4,V2,V3 ; add
vst V4,Ry ; store the result
```

- Requires 6 instructions vs. almost 250 for RISC V
Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- RV64V functional units consume one element per clock cycle
  - Execution time is approximately the vector length

Convoy
- Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards can be in the same convey via chaining

Chaining (within a convoy)
- Allows a vector operation to start as soon as the individual elements of its vector source operand become available

Chime
- Unit of time to execute one convoy
- $m$ convoys executes in $m$ chimes
- For vector length of $n$, a chime $\approx n$ clock cycles
**Example**

```
Example

vld V1,Rx ;load vector X
vmul.vs V2,V1,F0 ;vector-scalar multiply
vld V3,Ry ;load vector Y
vadd.vv V4,V2,V3 ;add two vectors
vst V4,Ry ;store the sum

Convoys:
1 vld vmul.vs
2 vld vadd.vv
3 vst

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5
For 32 element vectors, requires 32 x 3 = 96 clock cycles
```
Multiple Lanes

- Element $n$ of vector register $A$ is “hardwired” to element $n$ of vector register $B$
- Allows for multiple hardware lanes

Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum vector length (MVL):

```
low = 0;
VL = (n % MVL); /* find odd-size piece using modulo op % */
for (i = 0; j <= (n/MVL); j++) { /*outer loop*/
    for (i = low; i < (low+VL); i++) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i]; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```
Vector Mask Registers

- Consider the loop:
  
  ```
  for (i = 0; i < 32; i=i+1)
    if (X[i] != 0)
      X[i] = X[i] – Y[i];
  ```

- Use vector mask register, VM, to “disable” elements:
  
  ```
  vsetpcfgi 1 ;Enable 1 predicate register
  vld v0,Rx ;load vector X into V0
  vld v1,Ry ;load vector Y
  fild F0,#0 ;load FP zero into F0
  vpne p0, v0, F0 ;sets p0(i) to 1 if v0(i) != F0
  vsub.vv v0, v0, v1 ;subtract under vector mask
  vst v0,Rx ;store the result in X
  vdisable ; disable predicate registers
  ```

- GFLOPS rate decreases!

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory

- Ex: Assume the latency of a memory bank is 12 cycles
  
  - If 32 interleaved memory banks,
    - Can load a 32 element vector in 12+32= 44 cycles
  
  - If 16 interleaved memory banks,
    - Can load a 32 element vector in 44 cycles
  
  - If 8 interleaved memory banks,
    - The bank 0 delivers 4 elements at cycles 12, 24, 36 and 48
    - Bank 7 delivers 4 elements at cycles 20, 32, 44 and 56
    - Can load a 32 element vector in 56 cycles
### Stride

- **Consider:**
  ```c
  for (i = 0; i < 32; i=i+1)
      for (j = 0; j < 32; j=j+1) {
          A[i][j] = 0.0;
          for (k = 0; k < 32; k=k+1)
      }
  ```

- **Must vectorize multiplication of rows of B with columns of D**
- **Use** `vlds` (vector load with stride) **to load the columns of a matrix into a vector.**
  - `vlds v, (R1, R2) \rightarrow v[i] = address(R1+i R2)`
- **Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:**
  - `#banks / LCM(stride,#banks) < bank busy time`

### Scatter-Gather

- **Consider:**
  ```c
  for (i = 0; i < n; i=i+1)
      A[K[i]] = A[K[i]] + C[M[i]];
  ```

- **Use** `vldx` (load using index vector) **, and** `vstx`:
  - `vldx v1, (R1, v2) \rightarrow v1[i] = address(R1+v2[i])`
  - `vldx Va, (Ra,Vk) ;load A[K[]] -- Gather`
  - `vld Vm, Rm ;load M`
  - `vldx Vc, (Rc,Vm) ;load C[M[]]`
  - `vadd.vv Va, Va, Vc ;add them`
  - `vstx Va,(Ra,Vk) ;store A[K[]] -- Scatter`
**SIMD Extensions (section 4.3)**

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder

- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

**SIMD Implementations**

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (AVX) (2010)
    - Four 64-bit integer/fp ops

- Operands must be consecutive and aligned memory locations
Example SIMD Code

- Assume 64-bit architecture (8 bytes) and 4-SIMD operations
- Example Y[i] = Y[i] + a * x[i], i = 1,…,32:

  ```
  fld F0,a ; load scalar a
  splat.4D F0, F0 ; make 4 copies of F0 in F0, F1, F2 and F3
  addi R1,Rx,#256 ; last address to load (32 * 8)
  L: fld.4D F4,0[Rx] ; load X[i], X[i+1], X[i+2], X[i+3] to F4, F5, F6 and F7
  fmul.4D F4,F4,F0 ; a * X[i], a * X[i+1], a * X[i+2], a * X[i+3]
  fld.4D F8,0[Ry] ; load Y[i], Y[i+1], Y[i+2], Y[i+3] to F8, F9, F10 and F11
  fadd.4D F8,F8,F4 ; a * X[i] + Y[i], ..., a * X[i+3] + Y[i+3]
  fst.4D 0[Ry],F8 ; store into Y[i], Y[i+1], Y[i+2], Y[i+3]
  DADDIU Rx,Rx,#32 ; increment index to X (by 32 bytes – 4 words)
  DADDIU Ry,Ry,#32 ; increment index to Y (by 32 bytes – 4 words)
  bne Rx, R1, L ; will iterate 8 times, 4 Y's computed in each iteration
  ```

Loop-Level Parallelism (Sec. 4.5)

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

  ```
  for (i=0 ; i<1000 ; i++)
  x[i] = x[i] + s; // No loop-carried dependence
  ```

  ```
  for (i=2 ; i<1000 ; i++)
  x[i] = x[i-1] + x[i-2]; // loop-carried dependence
  ```
Loop-Level Parallelism

- Example 2:
  
  \[
  \text{for (i=0; i<100; i=i+1) \{}
  \]
  
  \[
  A[i+1] = A[i] + C[i]; /* S1 */
  \]
  
  \[
  B[i+1] = B[i] + A[i+1]; /* S2 */
  \]
  
  \[
  \} \]
  
  - S1 and S2 use values computed in previous iteration (bad for parallelism)
  - S2 uses value computed by S1 in same iteration
  - Iterations \(i, i+1, i+2, \ldots\) of the loop cannot be executed in parallel (non-parallelisable loop).

- Example 3:
  
  \[
  \text{for (i=0; i<100; i=i+1) \{}
  \]
  
  \[
  A[i] = A[i] + B[i]; /* S1 */
  \]
  
  \[
  B[i+1] = C[i] + D[i]; /* S2 */
  \]
  
  \[
  \} \]
  
  - S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
  - May transform to a parallel loop:
    
    \[
    \text{A[0] = A[0] + B[0];}
    \]
    
    \[
    \text{for (i=0; i<99; i=i+1) do in parallel \{} \]
    
    \[
    \text{B[i+1] = C[i] + D[i];}
    \]
    
    \[
    \text{A[i+1] = A[i+1] + B[i+1];}
    \]
    
    \[
    \}\]
    
    \[
    \text{B[100] = C[99] + D[99];}
    \]
Loop-Level Parallelism

- Example 4:
  for (i=0; i<100; i=i+1) {
    A[i] = B[i] + C[i];
    D[i] = A[i] * E[i];
  }

- Example 5:
  for (i=1; i<100; i=i+1) {
    Y[i] = Y[i-1] + Y[i];
  }

- Example 6:
  for (i=0; i<100; i=i+2) {
    A[2 * i + 1] = B[i] + C[i];
  }

Example 6 is a special case of the following:
  for (i=1; i<100; i=i+1) {
    A[a * i + b] = ....
    D[i] = A[c * i + d] + ....
  }

No loop carried dependences
Has loop carried dependences
Finding dependencies

- Assume indices are affine:
  - \( a \ast i + b \) (\( i \) is loop index)

- Assume:
  - Store to \( a \ast i + b \) and load from \( c \ast i + d \)
  - \( i \) is in the range from \( m \) to \( n \)
  - Dependence exists if:
    - There exists \( j, k \) such that \( m \leq j \leq n, m \leq k \leq n \)
    - Store to \( a \ast j + b \), load from \( c \ast k + d \), and \( a \ast j + b = c \ast k + d \)
  - GCD test (sufficient but not necessary):
    - A dependence does not exist if \( \text{GCD}(c, a) \) does not evenly divide \((d - b)\)

Finding dependencies

- Example 2:
  ```
  for (i=0; i<100; i=i+1) {
      Y[i] = X[i] / c; /* S1 */
      X[i] = X[i] + c; /* S2 */
      Z[i] = Y[i] + c; /* S3 */
      Y[i] = c - Y[i]; /* S4 */
  }
  ```

  - Watch for anti-dependencies and output dependencies
Assume 8 processors, how can you speed up the following loops?:

1. \[ \text{sum} = 0; \]
2. \[ \text{for } (i=0; i<16; i=i+1) \]
3. \[ \text{sum} += x[i]; \]
4. \[ \text{sum} = 0; \]
5. \[ \text{for } (i=0; i<400; i=i+1) \]
6. \[ \text{sum} += x[i]; \]
7. \[ \text{sum} = 0; \]
8. \[ \text{for } (i=0; i<16; i=i+1) \]
9. \[ x[i] = x[i-1] + x[i]; \]
10. \[ \text{for } (i=1; i<400; i=i+1) \]
11. \[ x[i] = x[i-1] + x[i]; \]

**Prefix computation**

```plaintext
half = 1
P0

half = 2
P0
P1

half = 4
P0
P1
P2
P3

half = 8
P0
P1
P2
P3
P4
P5
P6
P7
```

**Reductions**

```plaintext
half = 8; /* half = n/2 */
Repeat
{    if (Pid < half) x[Pid] = x[Pid] + x[Pid + half];
    half = half / 2;
    synchronize; /* a barrier */
} until (half = 0)
```