高等计算机系统结构
向量处理器
Vector Processor/
SIMD/GPU
2011.5.14
指令内部并行：Intra-instruction

• To achieve high performance
  – Memory latency and bandwidth
  – Reduce Instruction memory bandwidth Requirement
  – Intra-instruction Parallelism
    » CISC: dynamically translating to RISC-like u-ops
    » SIMD: multiple data operations per instruction
      • Data-level parallelism (DLP)
  – Code Density: Improving memory bandwidth efficiency

• Microprocessor Performance:

\[
\text{performance} = \left( \frac{1}{I_c} \times \frac{1}{\text{OPI}} \right) \times \text{OPC} \times f
\]
SIMD 簡介

• 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 并行

• 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!
向量体系结构 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
VMIPS

• Example architecture: VMIPS
  – Loosely based on Cray-1
  – Vector registers
    » Each register holds a 64-element, 64 bits/element vector
    » 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
Vector Programming Model

Scalar Registers

*v0, r0, r15*

Vector Registers

*v0, v1, v2, v3, v15*

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
VMIPS Instructions

• ADDVV.D: add two vectors
• ADDVS.D: add vector to a scalar
• LV/SV: vector load and vector store from address

• Example: DAXPY ( Y = a * X + Y )

L.D F0,a ; load scalar a
LV V1,Rx ; load vector X
MULVS.D V2,V1,F0 ; vector-scalar multiply
LV V3,Ry ; load vector Y
ADDVV V4,V2,V3 ; add
SV Ry,V4 ; store the result

• Requires 6 instructions vs. almost 600 for MIPS
### Vector Code Example

<table>
<thead>
<tr>
<th>C code</th>
<th>Scalar Code</th>
<th>Vector Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;64; i++) C[i] = A[i] + B[i];</td>
<td>LI R4, 64</td>
<td>LI VLR, 64</td>
</tr>
<tr>
<td></td>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L.D F0, 0(R1)</td>
<td>LV V1, R1</td>
</tr>
<tr>
<td></td>
<td>L.D F2, 0(R2)</td>
<td>LV V2, R2</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4, F2, F0</td>
<td>ADDVV V3, V1, V2</td>
</tr>
<tr>
<td></td>
<td>S.D F4, 0(R3)</td>
<td>SV V3, R3</td>
</tr>
<tr>
<td></td>
<td>DADDIU R1, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R2, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R3, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSUBIU R4, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNEZ R4, loop</td>
<td></td>
</tr>
</tbody>
</table>
Vector Execution Time

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

• VMIPS functional units consume one element per clock cycle
  – Execution time is approximately the vector length

• Convey
  – Set of vector instructions that could potentially execute together
Vector Instruction Execution

**ADDV C, A, B**

**Execution using one pipelined functional unit**


**Execution using four pipelined functional units**


- A[22] B[22]

- A[27] B[27]
Chimes

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

- **Chaining**
  - 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 convey
  - $m$ conveys executes in $m$ chimes
  - For vector length of $n$, requires $m \times n$ clock cycles
Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1

\[
\begin{align*}
&\text{LV } v_1 \\
&MULV v_3, v_1, v_2 \\
&ADDV v_5, v_3, v_4
\end{align*}
\]
Vector Chaining Advantage

• Without chaining, must wait for last element of result to be written before starting dependent instruction

• With chaining, can start dependent instruction as soon as first result appears
Vector Instruction Parallelism

Can overlap execution of multiple vector instructions
– example machine has 32 elements per vector register and 8 lanes

Instruction issue

Complete 24 operations/cycle while issuing 1 short instruction/cycle
Challenges

• Start up time
  – Latency of vector functional unit
  – Assume the same as Cray-1
    » Floating-point add => 6 clock cycles
    » Floating-point multiply => 7 clock cycles
    » Floating-point divide => 20 clock cycles
    » Vector load => 12 clock cycles

• Improvements:
  – > 1 element per clock cycle
  – Non-64 wide vectors
  – IF statements in vector code
  – Memory system optimizations to support vector processors
  – Multiple dimensional matrices
  – Sparse matrices
  – Programming a vector computer
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)
Multiple Lanes

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

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

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

\[ \text{for } (i=0; i < N; i++) \]
\[ C[i] = A[i] + B[i]; \]

Scalar Sequential Code

Iter. 1
- load
- load
- add
- store

Iter. 2
- load
- load
- add
- store

Vectorized Code

Iter. 1
- load
- load
- add
- store

Iter. 2
- load
- load
- add
- store

Vector Instruction

Vectorization is a massive compile-time reordering of operation sequencing \[ \Rightarrow \] requires extensive loop dependence analysis
Vector Length Register

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

```c
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*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*/
}
```

![Diagram of vector length distribution](image)
Vector Stripmining

Problem: Vector registers have finite length
Solution: Break loops into pieces that fit in registers, “Stripmining”

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

   +
  +  +  +
A B C

64 elements

   +
  +  +  +
Remainder

   +
  +  +  +
Remainder

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

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?
Vector Mask Registers

• Consider:

\[
\text{for } (i = 0; i < 64; i=i+1) \\
\text{if } (X[i] \neq 0) \\
\quad X[i] = X[i] - Y[i];
\]

• Use vector mask register to “disable” elements:

- \( \text{LV V1,Rx} \); load vector \( X \) into \( V1 \)
- \( \text{LV V2,Ry} \); load vector \( Y \)
- \( \text{L.D F0,#0} \); load FP zero into \( F0 \)
- \( \text{SNEVS.D V1,F0} \); sets \( VM(i) \) to 1 if \( V1(i) \neq F0 \)
- \( \text{SUBVV.D V1,V1,V2} \); subtract under vector mask
- \( \text{SV Rx,V1} \); store the result in \( X \)

• GFLOPS rate decreases!
Masked Vector Instructions

Simple Implementation
- execute all N operations, turn off result writeback according to mask

Density-Time Implementation
- scan mask vector and only execute elements with non-zero masks


M[1]=1  C[1]
M[0]=0  C[0]
Write Enable  Write data port


M[1]=1  C[1]
M[0]=0  C[0]
Write data port

Write data port
Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```plaintext
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector mask (or flag) registers
- vector version of predicate registers, 1 bit per element

...and maskable vector instructions
- vector operation becomes NOP at elements where mask bit is clear

Code example:

```plaintext
CVM       # Turn on all elements
LV vA, rA  # Load entire A vector
SGTVS.D vA, F0  # Set bits in mask register where A>0
LV vA, rB  # Load B vector into A under mask
SV vA, rA  # Store A back to memory under mask
```
Vector Reductions

Problem: Loop-carried dependence on reduction variables

\[
\text{sum} = 0; \\
\text{for} \ (i=0; i<N; i++) \\
\quad \text{sum} += A[i]; \quad \# \text{Loop-carried dependence on sum}
\]

Solution: Re-associate operations if possible, use binary tree to perform reduction

\[
\text{# Rearrange as:} \\
\text{sum}[0:VL-1] = 0 \quad \# \text{Vector of VL partial sums} \\
\text{for}(i=0; i<N; i+=VL) \quad \# \text{Stripmine VL-sized chunks} \\
\quad \text{sum}[0:VL-1] += A[i:i+VL-1]; \quad \# \text{Vector sum} \\
\text{# Now have VL partial sums in one vector register} \\
\text{do} \ { \text{\{}} \\
\quad \text{VL} = \text{VL}/2; \quad \# \text{Halve vector length} \\
\quad \text{sum}[0:VL-1] += \text{sum}[VL:2*VL-1] \quad \# \text{Halve no. of partials} \\
\text{\}} \text{ while (VL>1)}
\]
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

• Example:
  – 32 processors, each generating 4 loads and 2 stores/cycle
  – Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  – How many memory banks needed?
Interleaved Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Time before bank ready to accept next request

![Diagram of Interleaved Vector Memory System]

- **Vector Registers**
- **Memory Banks**
- **Address Generator**
- **Base**
- **Stride**
Stride

• Consider:
  for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
      A[i][j] = 0.0;
      for (k = 0; k < 100; k=k+1)
    }

• Must vectorize multiplication of rows of B with columns of D
• Use non-unit stride
• 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:
  for (i = 0; i < n; i=i+1)
    $A[K[i]] = A[K[i]] + C[M[i]]$;

• Use index vector:
  \begin{align*}
  &LV \quad Vk, Rk \quad ;\text{load } K \\
  &LVI \quad Va, (Ra+Vk) \quad ;\text{load } A[K[]] \\
  &LV \quad Vm, Rm \quad ;\text{load } M \\
  &LVI \quad Vc, (Rc+Vm) \quad ;\text{load } C[M[]] \\
  &ADDVV.D \quad Va, Va, Vc \quad ;\text{add them} \\
  &SVI \quad (Ra+Vk), Va \quad ;\text{store } A[K[]]
  \end{align*}
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

\[
\text{for (i=0; i}<N; i++)}
\]
\[
A[i] = B[i] + C[D[i]]
\]

Indexed load instruction (\textit{Gather})

\[
\text{LV vD, rD} \quad \# \text{ Load indices in D vector}
\]
\[
\text{LVI vC, rC, vD} \quad \# \text{ Load indirect from rC base}
\]
\[
\text{LV vB, rB} \quad \# \text{ Load B vector}
\]
\[
\text{ADDV.D vA,vB,vC} \quad \# \text{ Do add}
\]
\[
\text{SV vA, rA} \quad \# \text{ Store result}
\]
Vector Scatter/Gather

Scatter example:

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

Is following a correct translation?

```assembly
LV vB, rB       # Load indices in B vector
LVI vA, rA, vB  # Gather initial A values
ADDV vA, vA, 1  # Increment
SVI vA, rA, vB  # Scatter incremented values
```
SIMD Extensions

• 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 (2010)
    » Four 64-bit integer/fp ops

- Operands must be consecutive and aligned memory locations
Multimedia Extensions (aka SIMD extensions)

<table>
<thead>
<tr>
<th></th>
<th>64b</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b</td>
<td></td>
</tr>
<tr>
<td>16b</td>
<td>16b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
</tr>
</tbody>
</table>

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - This concept first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b
  - Newer designs have 128-bit registers (PowerPC Altivec, Intel SSE2/3/4)
- Single instruction operates on all elements within register

4x16b adds
Multimedia Extensions versus Vectors

• Limited instruction set:
  – no vector length control
  – no strided load/store or scatter/gather
  – unit-stride loads must be aligned to 64/128-bit boundary

• Limited vector register length:
  – requires superscalar dispatch to keep multiply/add/load units busy
  – loop unrolling to hide latencies increases register pressure

• Trend towards fuller vector support in microprocessors
  – Better support for misaligned memory accesses
  – Support of double-precision (64-bit floating-point)
  – New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)
Roofline Performance Model

• Basic idea:
  – Plot peak floating-point throughput as a function of arithmetic intensity
  – Ties together floating-point performance and memory performance for a target machine

• Arithmetic intensity
  – Floating-point operations per byte read
Examples

- Attainable GFLOPs/sec Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)
Graphical Processing Units

• Given the hardware invested to do graphics well, how can be supplement it to improve performance of a wider range of applications?

• Basic idea:
  – Heterogeneous execution model
    » CPU is the host, GPU is the device
  – Develop a C-like programming language for GPU
  – Unify all forms of GPU parallelism as CUDA thread
  – Programming model is “Single Instruction Multiple Thread”
Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS
NVIDIA GPU Architecture

• Similarities to vector machines:
  – Works well with data-level parallel problems
  – Scatter-gather transfers
  – Mask registers
  – Large register files

• Differences:
  – No scalar processor
  – Uses multithreading to hide memory latency
  – Has many functional units, as opposed to a few deeply pipelined units like a vector processor
Example

• Multiply two vectors of length 8192
  – Code that works over all elements is the grid
  – Thread blocks break this down into manageable sizes
    » 512 threads per block
  – SIMD instruction executes 32 elements at a time
  – Thus grid size = 16 blocks
  – Block is analogous to a strip-mined vector loop with vector length of 32
  – Block is assigned to a multithreaded SIMD processor by the thread block scheduler
  – Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors
Terminology

• **Threads of SIMD instructions**
  – Each has its own PC
  – Thread scheduler uses scoreboard to dispatch
  – No data dependencies between threads!
  – Keeps track of up to 48 threads of SIMD instructions
    » Hides memory latency

• Thread block scheduler schedules blocks to SIMD processors

• Within each SIMD processor:
  – 32 SIMD lanes
  – Wide and shallow compared to vector processors
Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    » 64 vector registers of 32 32-bit elements
    » 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers
NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - “Parallel Thread Execution (PTX)”
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

    shl.s32 R8, blockIdx, 9 ; Thread Block ID * Block size (512 or 29)
    add.s32 R8, R8, threadIdx ; R8 = i = my CUDA thread ID
    ld.global.f64 RD0, [X+R8] ; RD0 = X[i]
    ld.global.f64 RD2, [Y+R8] ; RD2 = Y[i]
    mul.f64 R0D, RD0, RD4 ; Product in RD0 = RD0 * RD4 (scalar a)
    add.f64 R0D, RD0, RD2 ; Sum in RD0 = RD0 + RD2 (Y[i])
    st.global.f64 [Y+R8], RD0 ; Y[i] = sum (X[i]*a + Y[i])
Conditional Branching

• Like vector architectures, GPU branch hardware uses internal masks

• Also uses
  – Branch synchronization stack
    » Entries consist of masks for each SIMD lane
    » I.e. which threads commit their results (all threads execute)
  – Instruction markers to manage when a branch diverges into multiple execution paths
    » Push on divergent branch
  – …and when paths converge
    » Act as barriers
    » Pops stack

• Per-thread-lane 1-bit predicate register, specified by programmer
Example

if (X[i] != 0)
    X[i] = X[i] - Y[i];
else X[i] = Z[i];

ld.global.f64 RD0, [X+R8] ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0 ; P1 is predicate register 1
@!P1, bra ELSE1, *Push ; Push old mask, set new mask
    bits
             ELSE1:  ld.global.f64 RD0, [Z+R8] ; RD0 = Z[i]
                     st.global.f64 [X+R8], RD0 ; X[i] = RD0
            ENDIF1:  <next instruction>, *Pop ; pop to restore old mask
                     st.global.f64 [X+R8], RD0 ; X[i] = RD0
                     @P1, bra ENDIF1 ; if P1 true, go to ENDIF1
             ENDIF1:  <next instruction>, *Pop ; pop to restore old mask
                     @P1, bra ELSE1, *Push ; Push old mask, set new mask
             ELSE1:
             ENDIF1:

Copyright © 2012, Elsevier Inc. All rights reserved.
NVIDIA GPU Memory Structures

• Each SIMD Lane has private section of off-chip DRAM
  – “Private memory”
  – Contains stack frame, spilling registers, and private variables

• Each multithreaded SIMD processor also has local memory
  – Shared by SIMD lanes / threads within a block

• Memory shared by SIMD processors is GPU Memory
  – Host can read and write GPU memory
Fermi Architecture Innovations

• Each SIMD processor has
  – Two SIMD thread schedulers, two instruction dispatch units
  – 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  – Thus, two threads of SIMD instructions are scheduled every two clock cycles

• Fast double precision
• Caches for GPU memory
• 64-bit addressing and unified address space
• Error correcting codes
• Faster context switching
• Faster atomic instructions
Loop-Level Parallelism

- 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=999; i>=0; i=i-1)
     x[i] = x[i] + s;
  ```

- No loop-carried dependence
Loop-Level Parallelism

• Example 2:
  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 by S1 in previous iteration
• S2 uses value computed by S1 in same iteration
Loop-Level Parallelism

• Example 3:
  
  ```c
  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

• Transform to:
  
  ```c
  A[0] = A[0] + B[0];
  for (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
  }
  B[100] = C[99] + D[99];
  ```
Loop-Level Parallelism

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

• Example 5:
  
  ```c
  for (i=1; i<100; i=i+1) {
      Y[i] = Y[i-1] + Y[i];
  }
  ```
Finding dependencies

• Assume indices are affine:
  – \(a \times i + b\) (i is loop index)

• Assume:
  – Store to \(a \times i + b\), then
  – Load from \(c \times i + d\)
  – \(i\) runs from \(m\) to \(n\)
  – Dependence exists if:
    » Given \(j, k\) such that \(m \leq j \leq n, m \leq k \leq n\)
    » Store to \(a \times j + b\), load from \(a \times k + d\), and \(a \times j + b = c \times k + d\)
Finding dependencies

• Generally cannot determine at compile time
• Test for absence of a dependence:
  – GCD test:
    » If a dependency exists, GCD(c,a) must evenly divide (d-b)

• Example:
  
```c
for (i=0; i<100; i=i+1) {
}
```
Finding dependencies

• Example 2:

```c
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 antidependencies and output dependencies
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 antidependencies and output dependencies
Reductions

- Reduction Operation:
  for (i=9999; i>=0; i=i-1)
      sum = sum + x[i] * y[i];

- Transform to…
  for (i=9999; i>=0; i=i-1)
      sum [i] = x[i] * y[i];
  for (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];

- Do on p processors:
  for (i=999; i>=0; i=i-1)
      finalsum[p] = finalsum[p] + sum[i+1000*p];

- Note: assumes associativity!
Parallel Architecture

(a) MIMD

(b) Vector-SIMD

Vector-SIMD Arithmetic Instructions  Vector-SIMD Memory Instructions  Architectural Vector Register with 4 Elements
Parallel Architecture

(c) Subword-SIMD
- Subword-SIMD Arithmetic Commands
- Full-Word Memory Commands
- Execution Resources w/ Subword Ops
- Architectural 64b Register w/ 8 Subwords

(d) SIMT
- Microthread Block w/ 4 Microthreads
- Memory

[Diagram showing parallel architecture with SIMD units and memory.]
Vector-Thread (ISCA’2011)
Magic/Complex Instruction

<table>
<thead>
<tr>
<th>Generality</th>
<th>Any application</th>
<th>Most applications</th>
<th>Single application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>2 - 8 instructions</td>
<td>1K – 100K instructions</td>
<td>10K – 10M instructions</td>
</tr>
</tbody>
</table>
BERERT (MICRO'11)
BETERT (MICRO’11)
Figure 1. Specialization Spectrum

a) Blackscholes  b) Streamcluster  c) 3x2 DySER  d) Blackscholes  e) Streamcluster
Complex Irregular Operation (HPCA’11)

C code:

```c
for (i=0; i<N; i++) {
    x = A[i];
    y = B[i];
    C[x] = D[y] + x + y + (x-1)*(y-1);
}
```