Chapter 4

Data-Level Parallelism

in

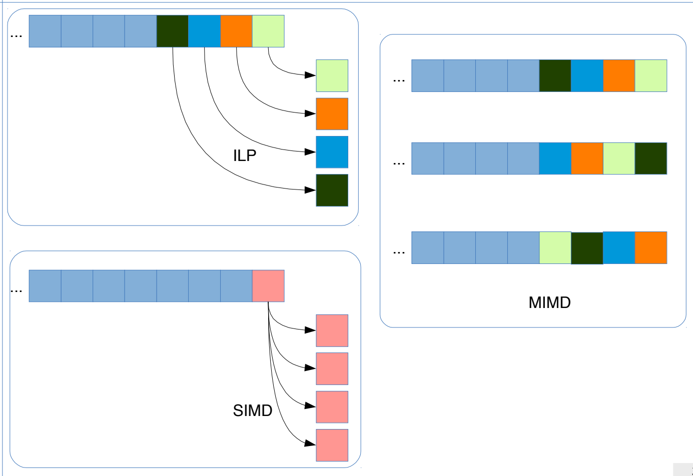
Vector, SIMD, and GPU

Architectures

Introduction: Focusing on SIMD

- 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
  - “Backwards compatibility”
- Three major implementations
  - Vector architectures
  - SIMD extensions
  - Graphics Processor Units (GPUs)

ILP vs. SIMD vs. MIMD



SIMD or MIMD?

- Based on past history, the book predicts that for x86
  - We will get two additional cores per chip per year
    - Linear growth
  - We will get twice as wide SIMD instructions every 4 years
    - Exponential growth
  - Potential speedup from SIMD to be twice that from MIMD!

Vector Architecture Basics

- The underlying idea:
  - The processor does everything on a set of numbers at a time
    - Read sets of data elements into “vector registers”
    - Operate on those vector registers
    - Disperse the results back into memory in sets
- Registers are controlled by compiler to...
  - ...hide memory latency
    - The price of latency paid only once for the first element of the vector
    - No need for sophisticated prefetcher
      - Always load consecutive elements from memory (unless gather-scatter)
  - ...leverage memory bandwidth

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: new instruction every cycle
  - 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 of MIPS
  - 32 floating-point registers of MIPS

How Vector Processors Work: Y=a*X+Y

```
L.D         F0, a
DADDIU  R4, Rx, #512
Loop:
L.D         F2, 0(Rx)
MUL.D       F2, F2, F0
L.D         F4, 0(Ry)
ADD.D       F4, F4, F2
S.D         F4, 9(Ry)
DADDIU  Rx, Rx, #8
DADDIU  Ry, Ry, #8
DSUB    R20, R4, Rx
BNEZ    Loop
```

600 instructions

```
L.D         F0, a
LV          V1, Rx
MULVS.D     V2, V1, F0
LV          V3, Ry
ADDVV.DV4, V2, V3
SV          V4, Ry
```

6 instructions, no loop

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- In VMIPS:
  - 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
    - No structural hazards
    - RAW hazards are avoided through chaining
      - In superscalar RISC it is called forwarding
  - Number of convoys gives an estimate of execution time

Chaining and Chimes

- Chaining
  - Like forwarding in superscalar RISC
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
  - Flexible chaining chaining with any other instruction in execution, not just the once sharing a register
- Chime
  - Unit of time to execute one convoy
  - m convoys executes in m chimes
  - For vector length of n, it is required m*n clock cycles
  - Ignores some processor specific overheads
    - Better at estimating time for longer vector processors
  - Chime is an underestimate

Example Calculation of Execution Time

```
LV          V1,Rx           ;load vector X
MULVS.D     V2,V1,F0                ;vector-scalar multiply
LV          V3,Ry           ;load vector Y
ADDVV.D     V4,V2,V3                ;add two vectors
SV          Ry,V4               ;store the sum
```

Convoys:

1 LV MULVS.D

2 LV ADDVV.D

3 SV

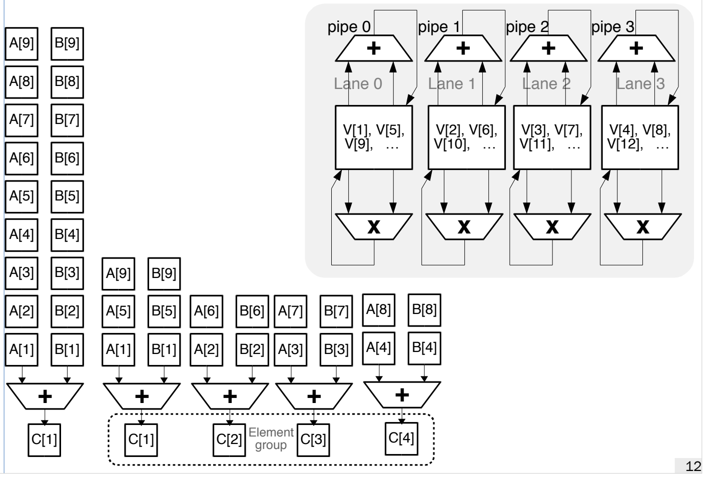
3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 64 x 3 = 192 clock cycles

Vector Execution Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add → 6 cycles
    - Floating-point multiply → 7 cycles
    - Floating-point divide → 20 cycles
    - Vector load → 12 cycles
- Needed improvements
  - > 1 element per clock cycle
    - Compare with CPI < 1
  - 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 Optimizations: Multiple Lanes



Arbitrary Length Loops with Vector-Length Register

- `for (i=0; i < N; ++i)  Y[i] = a * X[i] + Y[i]`
- What if N < 64 (=Maximum Vector Length)?
- Use a Vector-Length Registers
  - Load N into VLR
  - Vector instructions (loads, stores, arithmetic, …) use VLR for the maximum number of iterations
  - By default, VLR=64
  - If VLR<64 then the registers are not fully utilized
    - Vector startup time is a greater portion of the instruction
    - Use Amdahl's law to calculate the effect of diminishing vector length

Arbitrary Length Loops with Strip Mining

- `for (i=0; i < N; ++i)  Y[i] = a * X[i] + Y[i]`
- What if N > 64?
- Use strip mining to split the loop into smaller chunks

```
for (j = 0;  j < (N/64)*64;  j += 64) {//N – N%64
  for (i=0 ; i < 64; ++i)
    Y[j+i] = a * X[j+i] + Y[i]
}
```

- Clean-up code for N not divisible by 64

```
if (N % 64)
  for (i = 0; i < N%64; ++i)    // use VLR here
    Y[j+i] = a * X[j+i] + Y[i]
```

Handling Conditional Statements with Mask Regs.

-
  ```
  for (i=0 ; i < 64; ++i)
    if (X[i] != 0)
      X[i] = X[i] – Y[i]
  ```
- Values of X are not known until runtime
- The loop cannot be analyzed at compile time
- Conditional is the inherent part of the program logic
- Scientific codes are full of conditionals and hardware support is a must
- Vector mask register is used as to prohibit execution of vector instructions on some elements of the vector
  ```
  for (i = 0; i < 64; ++i)
    VectorMask[i] = (X[i] == 0)
  for (i = 0; i < 64; ++i)
    X[i] = X[i] – Y[i] only if VectorMask[i] != 0
  ```

Example: Vector Mask Register

-
  ```
  for (i=0 ; i<64; ++i)
    if (X[i] != 0)
      X[i] = X[i] – Y[i]
  ```
- Vector register utilization drops with the dropping count of 1's in mask register

-
  ```
  LV        V1, Rx
  LV        V2, Ry
  L.D       F0, #0
  SNEVS.DV1, F0
  SUBVV.DV1, V1, V2
  SV        V1, Rx
  ```

Strided Access and Multidimensional Arrays

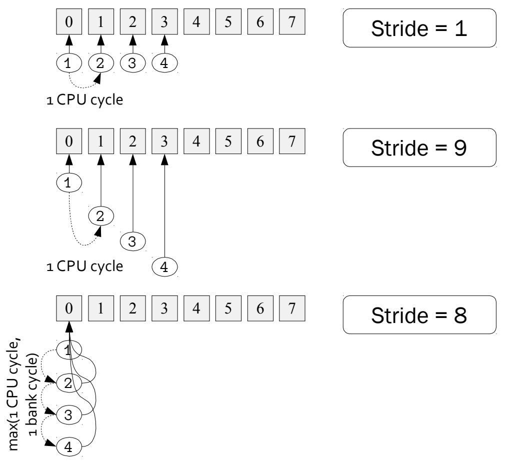
- Matrix-matrix multiply C= alpha * C + beta * A * B:
  ```
  for (i = 0; i < 64; ++i)
    for (j = 0; j < 64; ++j) {
      C[i, j] = alpha * C[i, j]
      for (k = 0; k < 64; ++k)
        C[i,j] = C[i,j] + beta * A[i, k] * B[k, j]
    }
  ```
- C[0,0] = A[0,0]*B[0,0]+A[0,1]*B[1,0]+A[0,2]*B[2,0]+…
- A is accessed by rows and B is accessed by columns
- Two most common storage formats:
  1. C: row-major (store matrix by rows)
  2. F: column-major (store matrix by columns)
- There is a need for non-unit stride for accessing memory
  - Best handled by memories with multiple banks
  - Stride: 64 elements * 8 bytes = 512 bytes

Strided Access and Memory Banks

- Special load instructions just for accessing strided data
  - LVWS=load vector with stride and SVWS
  - The stride (distance between elements) is loaded in GPR (general purpose register) just like vector address
- Multiple banks help with fast data access
  - Multiple loads and/or store can be active at the same time
  - Critical metric: bank busy time (minimum period of time between successive accesses to the same bank)
- For strided accesses, compute LCM (least common multiple) of stride and number of banks to see if bank conflict occurs

$$\frac{\#\text{banks}}{\text{LCM}(\text{stride}, \#\text{banks})} < \text{bank busy time cycles}$$

Strided Access Patterns



Example: Memory banks for Cray T90

- Design parameters
  - 32 processors
  - Each processor: 4 loads and 2 stores per cycle
  - Processor cycle time: 2.167 ns
  - Cycle time for SRAMs: 15 ns
- How many memory banks required to keep up with the processors?
  - 32 processors * 6 references = 192 references per cycle
  - SRAM busy time: 15/2.167 = 6.92 ~ 7 CPU cycles
  - Required cycles: 192 * 7 = 1344 memory banks
  - Original design had only 1024 memory banks.
  - A subsequent memory upgraded introduced synchronous SRAMs with half the cycle time

Sparse Matrices and Gather-Scatter Instruction

- Sparse vector add uses index vector:
```
for (i = 0 ; i<n; ++i)
  A[K[i]] = A[K[i]] + C[M[i]]
```
- Index vector is unknown during compilation
  - Index values may overlap
  - Hardware support: gather (LVI) scatter (SVI) instructions
- Assembly for VMIPS:
```
LV        Vk, Rk        load K
LVI       Va, (Ra, Vk)  load A[K[]]
LV        Vm, Rm        load M
LVI       Vc, (Rc, Vm)  load C[M[]]
ADDVV.D   Va, Va, Vc    add Va and Vc
SVI       (Ra+Vk), Va   store A[K[]]
```

Vector Processors and Compilers

- Vector machine vendors supplied good compilers to take advantage of the hardware
  - High profit margins allowed investment in compilers
  - Without well vectorized code, hardware cannot deliver high Gflop/s rates
- Vector compilers were…
  - Good at recognizing vectorization opportunities
  - Good at reporting vectorization process failures and successes
  - Good at taking hints from the programmer in form of vendor-specific language extensions such as pragmas

SIMD Extensions for Multimedia Processing

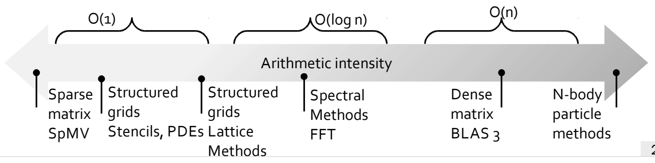
- Modern SIMD extensions in x86 date back to early desktop processing of multimedia
- Typical case of Amdahl's law:
  - Speedup the common case or the slowest component
    - Image processing: operates on 8 bit colors, 3 components, 1 alpha transparency
      - Typical pixel length: 32 bits
      - Typical operations: additions
    - Audio processing: 8 bit or 16 bit samples (Compact Disc)
- SIMD extensions versus vector processing
  - Fixed vector length encoded in instruction opcode for SIMD
  - Limited addressing modes in SIMD
  - No mask registers in early SIMD

SIMD Extensions History

- 1996
  - Intel MMX
    - 8 * 8-bit or 4 * 16-bit integer ops
- 1999
  - SSE (Streaming SIMD Extensions)
  - New 128-bit registers
    - 16 * 8-bit or 8 * 16-bit integer ops
    - Integer and single precision floating point
- 2001
  - SSE2
    - Double precision floating point
- 2004
  - SSE3
- 2007
  - SSE4
- 2010
  - AVX
  - Register width: 256-bits
  - FMA, permutations, shuffles
  - Possibilities for 512- and 1024-bit registers
- 2013
  - Intel Xeon Phi
  - 512-bit registers

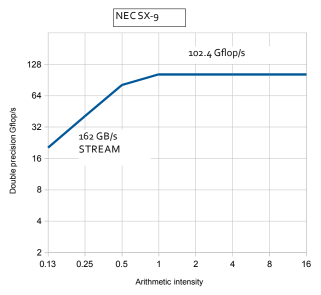
Roofline Performance Model

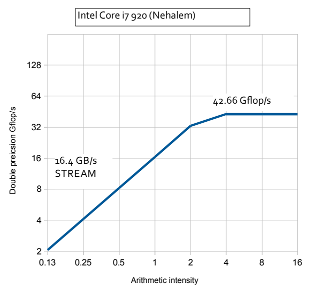
- From:
  - F1/2: A parameter to characterize memory and communication bottlenecks
  - by Roger W. Hockney and Ian J. Currington
- Application
  - Plot arithmetic intensity vs. peak floating point throughput
  - Benefit: correlates floating-point performance with memory bandwidth
- Arithmetic density:
  - Floating-point operations per bytes read



Roofline Examples

Attainable Gflop/s = min (Peak Memory B/W × Arithmetic intensity , Peak floating-point perf.)





From GPUs to GPGPUs to Hardware Accelerators

- CPUs are older than GPUs
- GPUs were specialized for graphics
- CPUs and GPUs were driven by different software interfaces
  - CPUs: systems languages and assembly
  - GPUs: OpenGL and DirectX
- Convergence happened around 2000
  - Realistic graphics requires continuous color models
    - Pixel colors represented by floating-point values not just 8-bit tuples
  - Special lighting effects cannot be fixed in hardware
    - Artists and designers drive realism of visualization effects
    - Accuracy and detail of visuals is game-dependent
    - This leads to programmable shaders, e.g. OpenGL shader language
    - Programmability is limited, based on graphics vocabulary, and hard to debug
      - Until CUDA happened and then OpenCL and the OpenACC

CUDA = Compute Unified Device Architecture

- Before CUDA…
  - Cg – C for graphics
  - Brook
  - Shader languages
    - No IEEE compliance
  - …
- CUDA includes
  - Programming language
    - C++ syntax
    - Pragmas and extra syntax
  - Programming model
    - Heterogeneous execution
      - Host and device
    - SIMT Single Instruction Multiple Thread
      - Thread: an abstraction of (any) parallelism
  - Programming interface
    - Initialization
    - Memory management
    - Data communication
    - Synchronization
    - Atomics

Example CUDA Program: DAXPY

```
void daxpy(int n, double a, double *x, double y) {
   for (int i = 0; i < n; ++i) y[i] = a*x[i] + y[i]
} // invocation: daxpy( 1024, 2.0, x, y )
__device__
void daxpy(int n, double a, double *x, double *y) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
} // invocation:
__host__
int nblocks = (n+255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
```

$\left\lceil \frac{n}{256} \right\rceil$

GPUs vs. Vector Processors

- Similarities
  - 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

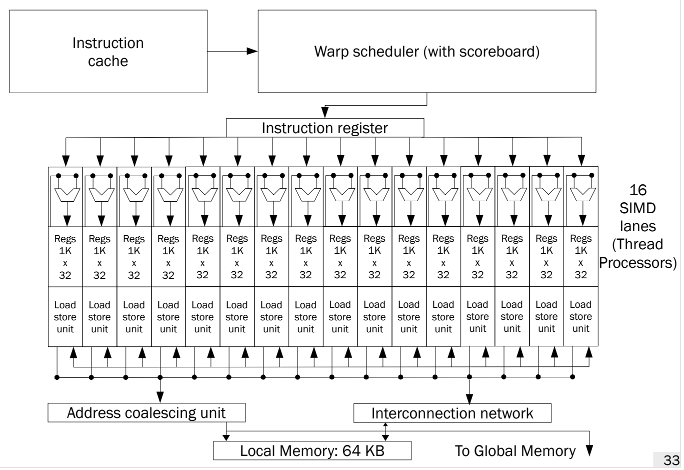
Programming Model

- Device(s) execute kernels
- Parallelism (SIMT) expressed as <u>CUDA Thread</u>
  - Multithreading
  - MIMD
  - SIMD
  - ILP
- Threads are managed by GPU hardware (not OS, not user)
- Threads executed in groups of 32 threads = thread blocks
- Block of threads executed on multithreaded SIMD processor
- Blocks of threads organized into a grid
- By comparison, 8-core Sandy Bridge with AVX has 64 GPU cores

Example: Multiply Two Vectors of Length 8192

- Programmer provides CUDA code for the entire <u>grid</u>
  - This is the equivalent of a vectorized loop
- The grid is composed of <u>thread blocks</u>
  - This is the equivalent of the body of loop
  - Up to 512 elements per thread block
  - Total thread blocks: 8192/512=16
- SIMD instruction executes 32 elements at a time
- Block is equivalent of a strip-mined vector loop with vector length 32
- Each block is assigned to a multithreaded SIMD processor by the thread block scheduler
- Fermi: 7-15 multithreaded SIMD processors
- Kepler: 7-15

Simplified Block Diagram of GPU



Levels of Scheduling and Threads

- Grid of thread blocks run on multithreaded SIMD Processors
  - Fermi: 7, 11, 14, 15, …
  - Kepler: 7, 8, …, 15, …
  - Each multithreaded SIMD processor
    - Has 32 SIMD lanes
    - Is wide and shallow compared to vector processors
- Thread block scheduler schedules thread blocks (vectorized loop bodies) to multithreaded SIMD processors
  - Ensures local memory has the needed data
- Threads of SIMD instructions
  - Machine object that is created, managed, scheduled, executed
  - Like traditional thread but with SIMD instructions only
  - Each thread has its own Program Counter
  - Thread scheduler uses scoreboard to dispatch ready threads
  - No data dependences between threads (only instructions)

Fermi GPU

- 16 physical SIMD lanes
- Each lane contains 2048 registers (32-bit)
- Total of 32768 registers!
- Each SIMD thread limited to 64 registers
- But each SIMD thread has access to vector-like registers
  - 64 registers, 32 elements each, 32 bits per element

(Gaming) Fermi vs. (Gaming) Kepler

- NVIDIA GeForce GTX 580
  - 512 SPs
  - 40 nm process
  - 192.4 GB/s memory b/w
  - 244 W TDP
  - 1/8 64-bit performance
  - 3 billion transistors
  - Die size: 100%
- NVIDIA GeForce GTX 680
  - 1536 SPs
  - 28 nm process
  - 192.2 GB/s memory b/w
  - 195 W TDP
  - 1/24 64-bit performance
  - 3.5 billion transistors
  - Die size: 56%

NVIDIA's Instruction Set Architectures

- 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 2^9)
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
  - Vector processors rely more heavily on software for conditionals
- Per-thread-lane 1-bit predicate register, specified by programmer
- Also uses a stack structure with markers for fast unwinding
  - 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

GPU Predication 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
                               ; if P1 false, go to ELSE1

ld.global.f64   RD2, [Y+R8]             ; RD2 = Y[i]
sub.f64         RD0, RD0, RD2           ; Difference in RD0
st.global.f64   [X+R8], RD0             ; X[i] = RD0
@P1, bra ELSE1:  ENDIF1, *Comp          ; complement mask bits
                                        ; if P1 true, go to ENDIF1

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
```

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
  - Fully pipelined and IEEE compliant
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

Loop-Level Parallelism in Code

- 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)
  x[i] = x[i] + s;
```

  - No loop-carried dependence

Example: Recognizing Dependences in Code

- Example 2:

```
for (i=0; i<100; ++i) {
   A[i+1] = A[i] + C[i];    /* S1 */
   B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- A, B are assumed disjoined arrays (as in Fortran)
- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration
- What is the value of A[0]?

Example: Removing Dependence

- Consider:

```
for (i=0; i<100; ++i) {            A[0]+= B[0];
   A[i] = A[i] + B[i];   /* S1 */  B[1]=C[0]+D[0];
   B[i+1] = C[i] + D[i];/* S2 */  A[1]+=B[1];
}                                  B[2]=C[1]+D[1];
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; ++i) {
   B[i+1] = C[i] + D[i];
   A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Finding Dependences: Theoretical Analysis

- Assume indices are affine:
  - Array index for storing data = a*i+b (i is loop index)
  - Array index for loading data = c*i+d (i is loop index)
- Assume:
  - Store to a*i+b, then
  - Load from c*i+d
  - i runs from m to n
  - Dependence exists if:
    - Given j,k such that m≤j≤n,m≤k≤n
    - Store to a*j+b, load from a*k+d, and a*j+b=c*k+d
- Generally cannot determine at compile time
- Instead, test for absence of a dependence:
  - GCD test:
  - If a dependency exists, GCD(c,a) must evenly divide (d-b)

Example for GCD Test

- Determine dependence for the following loop:

```
for (i=0; i<100; ++i) {
   X[2*i+3] = X[2*i] * 5.0;
}
```

- From the previous slide: a=2, b=3, c=2, d=0
- GCD(a,c) = 2 and d-b=-3
- 2 does not divide -3 hence no dependences exist

Weaknesses of the GCD Test

- GCD is sufficient to guarantee no dependences
  - But GCD is not a necessary condition
  - Even if GCD test succeeds there might not be dependence
  - Mostly due to loop bounds not being accounted for in GCD
- General dependence testing is NP-complete

Types of Dependences in Loops

-

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

- True dependences: S1 ⟶ S3, S1 ⟶ S4 because Y[i] but not loop-carried, so the loop is parallel
- Antidependences: S1 ⟶ S2 X[i], S3 ⟶ S4 Y[i]
- Output dependence: S1 ⟶ S4 Y[i]
-

```
for (i=0; i<100; ++i) {
   T[i] = X[i] / c;   /* Y renamed to T */
   X1[i] = X[i] + c; /* X renamed to X1 */
   Z[i] = T[i] + c;   /* Y renamed to T */
   Y[i] = c - T[i];   /* Y renamed to T */
}
```

- No copying of arrays necessary to remove dependences

Parallelizing Reductions

- Reduction operation:
  ```
  for (i=9999; i>=0; --i)
    sum = sum + x[i] * y[i];
  ```
- Transform to…
  ```
  for (i=9999; i>=0; --i)
    sum [i] = x[i] * y[i];

  for (i=9999; i>=0; --i)
    finalsum = finalsum + sum[i];
  ```
- Do on p processors (p between 0 and 9):
  ```
  for (i=999; i>=0; --i)
    finalsum[p] = finalsum[p] + sum[i+1000*p];
  ```
- Note: assumes associativity!