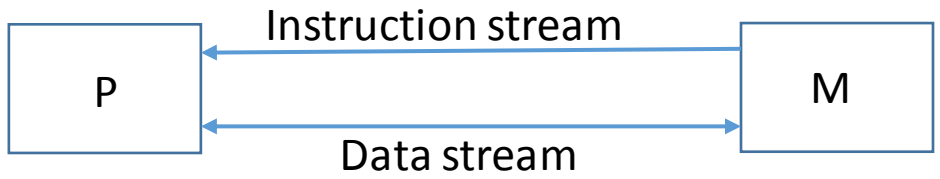
Module-2
CSEN 3104
Lecture 13

Dr. Debranjan Sarkar

SIMD Architecture

Flynn's classification



- Single Instruction Stream Single Data Stream (SISD)
  - Conventional machines with single CPU
  - Capable of only scalar arithmetic
  - $m_I = m_D = 1$

m_I : minimum number of active instruction streams
m_D : minimum number of active data streams

- Single Instruction Stream Multiple Data Stream (SIMD)
  - Single program control unit and many independent execution units
  - Example: Illiac IV
  - $m_I = 1, m_D > 1$
- Multiple Instruction Stream Single Data Stream (MISD)
  - Several CPUs process the same data using different programs
  - Fault tolerant computers
  - $m_I > 1, m_D = 1$
- Multiple Instruction Stream Multiple Data Stream (MIMD)
  - Computers with more than one CPU with the ability to execute several programs simultaneously
  - Multi Processors containing two or more CPUs that cooperate on common computational tasks
  - $m_I > 1, m_D > 1$

SIMD Array Processor

- A synchronous array of parallel processors
- Consists of multiple processing elements (PEs) under the supervision of one control unit (CU)
- Can handle single instruction and multiple data (SIMD) streams
- Specially designed to perform vector computations over matrices or arrays of data
- Two basic architectural organizations of SIMD computers
  - Array processors using distributed memories
  - Associative processors using content-addressable (or associative) memory
- Two configurations of Array Processors
  - Illiac IV (developed by the Illinois Automatic Computer team of University of Illinois)
  - Burroughs Scientific Processor (BSP)

Configuration of Illiac IV

- Show the block diagram
- N numbers of identical synchronized processing elements (PEs) which can concurrently do the same operation on different data
- All these PEs are under the control of one CU
- Each PE is basically an ALU with working registers and local memory (PEM) for the storage of distributed data
- The CU has its own main memory for the storage of programs
- The system and user programs are executed under the control of CU
- Scalar and control type instructions are executed in the CU
- Vector instructions are broadcast to the PEs for distributed execution
- Thus spatial parallelism is achieved through multiple arithmetic units (PEs)

Configuration of Illiac IV

- All the PEs perform the same function synchronously in a lock-step fashion under the command of the CU
- Vector operands are distributed to the PEMs before parallel execution in the array of PEs
- The distributed data can be into the PEMs from an external source via the system data bus or via the CU in a broadcast mode using the control bus
- Masking schemes are used to enable or disable each PE to participate in the execution of a vector instruction
- Data exchanges among the PEs are done via an interconnection network which performs all necessary data routing and manipulation functions
- This interconnection network is under the control of CU

Configuration of Illiac IV

- The Array Processor is normally interfaced to a host computer through the CU
- The host computer is a general purpose machine which serves as the operating manager of the entire system, consisting of the host and the processor array
- The functions of the host computer include resource management, peripheral and IO supervision
- The control unit of the processor array directly supervises the execution of programs, whereas
- The host machine performs the executive and IO functions with the outside world
- So, an array processor can be considered a back-end attached computer

SIMD Computer

- AN SIMD computer is characterized by a set of parameters:

  C = <N, F, I, M>

- N = Number of processing elements (PEs) in the system. For Illiac IV, N = 64
- F = a set of data routing functions provided by the interconnection network
- I = the set of machine instructions for scalar-vector, data routing, and network manipulation operation
- M = the set of masking schemes, where each mask partitions the set of PEs into two disjoint subsets of enabled PEs and disabled PEs
- This model provides a common basis for evaluating different SIMD machines

Components in a Processing Element (PE_i)

- Show figure
- Each PE_i has
  - its own memory PEM_i
  - A set of working registers and flags (A_i,B_i, C_i and S_i)
  - An Arithmetic and Logic Unit (ALU)
  - A local Index Register (I_i), an address register (D_i) and a data routing register (R_i)
  - The R_i of each PE_i is connected to the R_j of other PEs via the interconnection network
  - During data transfer among PEs, the contents of the R_i registers are transferred
  - The inputs and outputs of the R_i are totally isolated
  - The i^{th} PE is denoted by PE_i where the index i is the address of PE_i
  - Let the total number of PEs be $N = 2^m$, then $m = \log_2 N$ binary digits are needed to encode the address of a PE
  - The address register D_i is used to hold the m-bit address of the PE_i
  - Some array processors may use two routing registers – one for input and the other for output

Masking

- During each instruction cycle, each PE_i is either
  - in the active mode, or
  - in the inactive mode
- In case a PE_i is active, it executes the instruction, broadcast to it by the CU, otherwise it won't
- The masking schemes are used to specify the status flag S_i of PE_i
- $S_i = 1$ indicates an active PE_i and $S_i = 0$ indicates an inactive PE_i
- In the CU, there is a global
  - index register (I) and
  - N-bit masking register (M)
- The collection of S_i flags for i = 0, 1, 2, …., N-1 forms a status register S for all the PEs
- The bit patterns in registers M and S are exchangeable under the control of CU when masking is to be set

Thank you

Module-2
CSEN 3104
Lecture 14

Dr. Debranjan Sarkar

SIMD Architecture

Use of indexing to address the local memories in parallel at different local addresses

- Consider an array of n X n data elements:

$$A = \{A(i,j),\ 0 \leq i, j \leq (n-1)\}$$

- Elements of j^{th} column of A are stored in n consecutive locations of PEM_j [say from location 200 to location (200+n-1)] (assume $n \leq N$)
- We want to access the principal diagonal elements A(j,j) for j=0, 1 , …, (n-1) of the array A
- The CU must generate and broadcast an effective memory address 200
- The local index registers must be set to be $I_j = j$ for $j = 0, 1, \ldots, (n-1)$ in order to convert the global address 200 to local address $200 + I_j = 200 + j$ for each PEM_j
- Within each PE, there is a separate memory address register for holding these local addresses

Data Routing Mechanisms

- Execution of the following vector instruction in an array of N processing elements (PEs)
- The sum S(k) of the first k components in a vector $A = (A_0, A_{1, \ldots\ldots, } A_{n-1})$ is desired for each k from 0 to (n-1)
- We need to compute the following n summations:

$$S(k) = \sum_{i=0}^{k} A_i \qquad \text{for } k = 0, 1, \ldots\ldots, (n-1)$$

Data Routing Mechanisms

- These n vector summations can be computed recursively by going through the following (n-1) iterations:

$$S(0) = A_0$$

$$S(k) = S(k-1) + A_k \qquad \text{for } k = 1, 2, \ldots\ldots, (n-1)$$

- For n = 8, the above recursive summation is implemented in an array processor with N = 8 processing elements (PEs)
- $Log_2 n = 3$ steps are required
- Both data routing and PE masking are used
- Show diagram
- Initially each A_i, residing in PEM_i is moved to the R_i register in PE_i for i = 0,1,2,…,7

Data Routing Mechanisms

- In the first step, A_i is routed from R_i to R_{i+1} and added to A_{i+1} with the resulting sum $A_i + A_{i+1}$ in R_{i+1} for i = 0,1,2,…,6
- In step 2, the intermediate sums in R_i are routed to R_{i+2} for i = 0 to 5
- In step 3, the intermediate sums in R_i are routed to R_{i+4} for i = 0 to 3
- Thus, the final value of PE_k will be S(k) for k = 0,1,2,…,7

Data Routing Mechanisms

- In step 1, PE_7 is not involved in data routing (receiving but not transmitting)
- In step 2, PE_7 and PE_6 are not involved in data routing
- In step 3, PE_7, PE_6, PE_5 and PE_4 are not involved in data routing
- These unwanted PEs are masked off during the corresponding steps
- During the addition operations
  - PE_0 is disabled in step 1
  - PE_0 and PE_1 are made inactive in step 2
  - PE_0, PE_1, PE_2 and PE_3 are masked off in step 3
- The PEs that are masked off in each step depend on the operation (data-routing or addition)
- Thus the masking pattern keep changing in different operation cycles
- Masking and routing operation are much more complicated when the vector length n > N

Thank you

Module-2
CSEN 3104
Lecture 19
22/08/2019

Dr. Debranjan Sarkar

SIMD Algorithms

Matrix multiplication

Matrix Multiplication basics

- Let $A = [a_{ik}]$ and $B = [b_{kj}]$ be n x n matrices
- Product matrix $C = A \times B = [c_{ij}]$ of dimension n x n
- The elements of the product matrix C is related to elements of A and B by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj} \text{ for } 1 \leq i \leq n \text{ and } 1 \leq j \leq n$$

- There are n^3 cumulative multiplications to be performed.
- Cumulative multiplication refers to the linked multiply-add operation

  $c \leftarrow c + a \times b$.
- Addition is merged into the multiplication because the multiply is equivalent to multi-operand addition
- Unit time is considered as the time required to perform one cumulative multiplication

Matrix multiplication in SISD computer

```
For i = 1 to n Do
    For j = 1 to n Do
        Cij = 0                (Initialization)
        For k = 1 to n Do
            Cij = Cij + Aik . Bkj  (Scalar additive multiply)
        End of k loop
    End of j loop
End of i loop
```

- In a conventional SISD uniprocessor system, the n^3 cumulative multiplications are carried out by a serially coded program with 3 levels of DO loops corresponding to three indices to be used
- The time complexity of this sequential program is proportional to n^3

Matrix multiplication in SIMD computer

For i = 1 to n Do

  Par for k = 1 to n Do

    $C_{ik} = 0$ (Vector load)

  For j = 1 to n Do

    Par for k = 1 to n Do

      $C_{ik} = C_{ik} + A_{ij} . B_{jk}$ (Vector multiply)

  End of j loop

End of i loop

Matrix multiplication in SIMD computer

- There are n PEs
- The algorithm construct depends heavily on the memory allocations of the A, B, and C matrices in the PEMs
- Each row vector of the matrix is stored across the PEMs (Show figure)
- Column vectors are then stored within the same PEM
- This allows parallel access of all the elements in each row vector of the matrices
- First parallel do operation corresponds to vector load for initialization
- Other parallel do operation corresponds to vector multiply for the inner loop of additive multiplications
- The time complexity has been reduced to $O(n^2)$
- SIMD algorithm is n times faster than the SISD algorithm for matrix multiplication

Matrix multiplication in SIMD computer

- Vector load operation is performed to initialize the row vectors of matrix C, one row at a time
- For vector multiply operation, the same multiplier a_{ij} is broadcast from the CU to all the PEs to multiply all n elements of the i^{th} row vector of B
- In total, n^2 vector multiply operations are needed in the double loops
- Show table illustrating the successive contents of the C Array in memory
- Each vector multiply instruction implies n parallel scalar multiplications in each of the n^2 iterations

Sorting on a mesh-connected parallel computer

Parallel Sorting on mesh

- Sorting of $N = n^2$ elements on an $n \times n$ mesh-type processor array
- Architecture (show figure) is similar to Illiac IV with exceptions
  - No wraparound connections, i.e.,
  - PEs at the perimeter have 2 or 3 rather than 4 neighbours
  - This simplifies the array sorting algorithm
- Two time measures are required to estimate the time complexity of the algorithm:
  - Routing time (t_R) to move one data item from a PE to one of its neighbours
  - Comparison time (t_C) for one comparison step (conditional interchange on the contents of two registers in each PE

Parallel Sorting on mesh

- Concurrent data routing is allowed
- Upto N numbers of concurrent comparisons may be performed
- This means that a comparison-interchange step between two items in adjacent processors can be done in time $2t_R + t_c$ *(route left, compare, and route* right)
- A number of these comparison-interchange steps may be performed concurrently in time ($2t_R + t_c$) *if* they are all between distinct, vertically adjacent processors
- A mixture of horizontal and vertical comparison-interchanges will require at least ($4t_R + t_c$) time unit

Thank you

Module-2
CSEN 3104
Lecture 20
26/08/2019

Dr. Debranjan Sarkar

SIMD Algorithms

Sorting on a mesh-connected parallel computer

Parallel Sorting on mesh

- Sorting of $N = n^2$ elements on an $n \times n$ mesh-type processor array
- Architecture (show figure) is similar to Illiac IV with exceptions
  - No wraparound connections, i.e.,
  - PEs at the perimeter have 2 or 3 rather than 4 neighbours
  - This simplifies the array sorting algorithm
- Two time measures are required to estimate the time complexity of the algorithm:
  - Routing time (t_R) to move one data item from a PE to one of its neighbours
  - Comparison time (t_C) for one comparison step (conditional interchange on the contents of two registers in each PE

Parallel Sorting on mesh

- Concurrent data routing is allowed
- Upto N numbers of concurrent comparisons may be performed
- This means that a comparison-interchange step between two items in adjacent processors can be done in time $2t_R + t_c$ *(route left, compare, and route* right)
- A number of these comparison-interchange steps may be performed concurrently in time ($2t_R + t_c$) *if* they are all between distinct, vertically adjacent processors
- A mixture of horizontal and vertical comparison-interchanges will require at least ($4t_R + t_c$) time unit

Parallel Sorting on mesh

- The PEs may be indexed by a bijection from {1, 2, …., n} × {1, 2, …., n} to {0, 1, …, N - 1}, where $N = n^2$
- N elements of a linearly ordered set are initially loaded in the N PEs
- Sorting problem is defined as the problem of moving the j^{th} smallest element to the processor indexed by j for all j = 0, 1 …., N – 1
- Example:
- The elements (N=16, n = 4) to be sorted are initially loaded in the 4 X 4 array of PEs (Show Figure)
- Three ways of indexing the processors
  - *Row-major indexing*
  - *Shuffled row-major indexing*
  - *Snake-like row-major indexing*

Parallel Sorting on mesh

- *Row-major indexing (Show diagram)*
- *Shuffled row-major indexing (Show diagram)*
  - Note that this indexing is obtained by shuffling the binary representation of the row-major index
  - For example, the row-major index 5 has the binary representation 0101
  - Shuffling the bits gives 0011 which is 3
  - In general, the shuffled binary number, say, "abcdefgh" is "aebfcgdh“
- *Snake-like row-major indexing (Show diagram)*
  - Obtained from the row-major indexing by reversing the ordering in even rows
- The choice of a particular indexing scheme depends upon how the sorted elements will be used
- We are interested in designing algorithms which minimize the time spent in routing and comparing

Parallel Sorting on mesh

- For any index scheme, there are situations where the two elements initially loaded at the opposite corner PEs, have to be transposed during the sorting (Show Figure)
- This transposition needs at least 4(n - 1) routing steps
- This implies that no algorithm can sort n^2 elements in time less than *O(n)*
- Thus an O(n) sorting algorithm is considered optimal on a mesh of n^2 PEs
- We shall show one such optimal sorting algorithm on the mesh-connected PEs

Odd-even Transposition Sort

- Different Sorting algorithms
  - Bubble SortComputational Complexity: $O(n^2)$ in average case
  - Merge Sort Computational Complexity: O(n log n) in worst case
  - Quick Sort Computational Complexity: O(n log n) in average case
- These algorithms are not easily parallelizable
  - Because the operations depend on the result of the previous operations
- Odd-even Transposition sort (or Brick Sort) is suitable for parallel computers and the time complexity is reduced to O(n)
- Examples of Odd-Even Transposition Sort

Review of Batcher's odd-even merge sort

- Sort the first half of a list, and sort the second half separately
- Sort the odd-indexed entries (first, third, fifth, …) and the even-indexed entries (second, fourth, sixth, …) separately
- Make only one more comparison-switch per pair of keys to completely sort the list
- List of numbers: 2 7 6 3 9 4 1 8
- We wish to sort it from least to greatest
- If we sort the first and second halves separately we obtain: 2 3 6 7 / 1 4 8 9
- Sorting the odd-indexed keys (2, 6, 1, 8) we get (1 2 6 8)
- Sorting the even-indexed keys (3, 7, 4, 9) we get (3 4 7 9)
- Leaving them in odd and even places respectively yields: 1 3 2 4 6 7 8 9
- This list is now almost sorted
- Doing a comparison switch between the keys in positions (2 and 3), (4 and 5) and (6 and 7) will finish the sort

Review of Batcher's odd-even merge sort

- Normally, the length of the list is a power of 2 (Here $2^3 = 8$)
- Two sorted sequences are loaded on a set of linearly connected PEs (Show Figure)
- In the first stage, the odd-indexed elements are placed in the left and then the even-indexed elements are placed in the right. This is basically unshuffle (or inverse shuffle) operation
- In the second stage, the odd sequences and the even sequences are merged
- The third stage is basically a perfect shuffle operation
- The fourth and final stage is a comparison-interchange operation of even-indexed elements with the next element
- Note that the perfect shuffle can be achieved by using the triangular interchange pattern (show figure)
- Similarly, an inverted triangular interchange pattern will do the unshuffle.
- The double-headed arrows indicate interchanges

Thank you

Module-2
CSEN 3104
Lecture 21
27/08/2019

Dr. Debranjan Sarkar

SIMD Algorithms

Sorting on a mesh-connected parallel computer

Odd-even merge on a rectangular array of PEs

- Batcher's odd-even merge sort on a linear array can be generalized to a square array of PEs
- Let M(j,k) be the algorithm of merging two j-by-k/2 sorted adjacent subarrays to form a sorted j-by-k array, where j, k are powers of 2, and k > 1
- All the arrays are arranged in the snake-like row major ordering
- When j = 1 and k = 2, i.e., in case of M(1,2), a single comparison-interchange step is sufficient to sort two unit subarrays

Odd-even merge on a rectangular array of PEs

- Given two sorted columns of length j ≥ 2, M(j, 2) consists of the following steps:
- J1. Move all odds to the left column and all evens to the right. Time: $2t_R$
- J2. Use the "odd-even transposition sort" to sort each column

  Time: $j\ (2t_R + t_c)$
- J3. Interchange on even rows. Time: $2t_R$
- J4. One step of comparison-interchange (every "even" with the next "odd")

  Time: $2t_R + t_c$
- So total time required = $2t_R + j\ (2t_R + t_c) + 2t_R + (2t_R + t_c) = (6 + 2j)\ t_R + (1 + j)\ t_c$

- Show Figure to illustrate the algorithm M(j, 2) for j = 4

Odd-even merge on a rectangular array of PEs

- For j > 2 and k > 2, M(j, k) is defined recursively in the following way:
- M1. If j > 2, perform a single interchange step on even rows
  If j = 2, do nothing Time: $2t_R$
- M2. Unshuffle each row Time: $(k - 2)t_R$
- M3. Merge by calling M(j, k/2) on each half Time: T(j, k/2)
- M4. Shuffle each row Time: $(k - 2)t_R$
- M5. Interchange on even rows Time: $2t_R$
- M6. Comparison-interchange of adjacent elements
  (every "even" with the next "odd“) Time: $4t_R + t_c$

Odd-even merge on a rectangular array of PEs

- Steps M1 and M2 unshuffle the elements
- Step M3 recursively merges the "odd sequences“ and the "even sequences”
- Steps M4 and M5 shuffle the "odds" and "evens" together
- Step M5 performs the final comparison-interchange
- Show figure to illustrate the algorithm M(4, 4), where the two given sorted 4-by-2 subarrays are initially stored in 16 processors

Odd-even merge on a rectangular array of PEs

- Let T(j, k) be the time needed by M(j, k). Then we have
- $T(j, 2) = (2j + 6)t_R + (j + 1)t_C$ for $k = 2$
- $T(j, k) = (2k + 4)t_R + t_C + T(j, k/2)$ for $k > 2$
- By repeated substitution, we have the following time bound:

  $T(j, k) \leq (2j + 4k + 4\log_2 k)t_R + (j + \log_2 k)t_c$
- For n x n array of PEs, the M(n,n) sort algorithm can be done in T(n,n) time which is proportional to O(n):

  $T(n,n) = (6n + 4\log_2 n)t_R + (n + \log_2 n)t_c = O(n)\ [t_c \leq t_R]$
- A speedup of $O(\log_2 n)$ achieved over the best sorting algorithm (Quicksort), which takes $O(n\log_2 n)$ steps on a uniprocessor system (in the best case and in the average case)

Thank you