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:

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

- 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

$C_{ij} = 0$ (Initialization)

For k = 1 to n Do

$C_{ij} = C_{ij} + A_{ik} . B_{kj}$ (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 Sort Computational 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 $j \geq 2$, $M(j, 2)$ 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:

  $T(j, k) \le (2j + 4k + 4\log_2 k)t_R + (j + \log_2 k)t_c$
- 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):

  $T(n,n) = (6n + 4\log_2 n)t_R + (n + \log_2 n)t_c = O(n)$ $[t_c \le t_R]$
- 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