ELEN4020: Data Intensive Computing

Lynch Mwaniki 1043475
Madimetja Sethosa 1076467
Teboho Matsheke 1157717

25/02/2019

Multi-Dimensional Arrays

A common way to access elements in multidimensional arrays requires iterating through the dimensions using nested loops. The number of loops required is directly proportional to the number of dimensions of the array. This method allows elements to be accessed in a static array, but it is computationally expensive. This method would be difficult to implement when working with dynamic arrays and would also increase computation time.

In computer memory, arrays are stored as a contiguously. In the C programming language, arrays are stored in memory using row major order. The co-ordinates of the elements in an N dimensional array can be transformed to a 1-dimensional co-ordinate system. This allows for easy access to the elements by using a pointer, thus eliminating the need for nested loops.

3D Matrix Multiplication

In order to simplify the 3D matrix multiplication an expansion is made on the 2d matrix multiplication concept. The 3D matrices are represented as layers of 2d matrices. If a 3D matrix A is of size (N,N,N), it will consist of N 2d matrices with sizes (N,N).

rank3TensorMult function

The function takes two pointers to the two 3D matrices being multiplied, their *size* and *total number of elements*. It is assumed that the two matrices being multiplied are of the same size, thus there is no size checking function. Firstly, an array is created using the *total number of elements* to store the multiplication results. A **for** loop is used to traverse through the two 3D matrices to extract the N 2D layers. In each loop, the 2D layers are fed to the *rank2TensorMult()* function and the results are then pushed into the results array. The representation can be viewed from Figure 1.

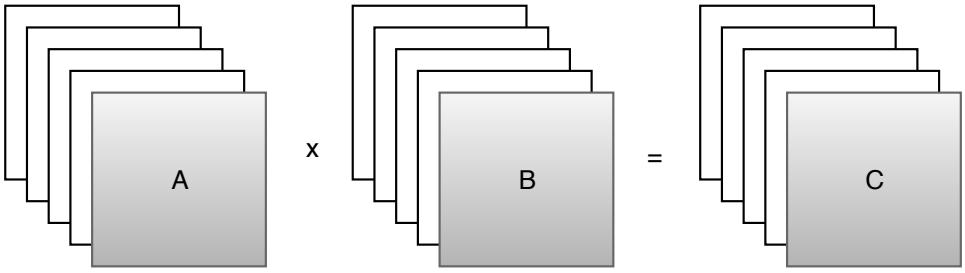

Figure 1: 3D matrix layer multiplication.

The *rank2TensorMult()* function uses one **for** loop to do the computations, this results in a total of two **for** loops for the *rank3TensorMult()* function. For more details about the function and other assisting functions, see Appendix A for the Pseudo code.

A Pseudo code

A.1 allocateMatrix

Algorithm 1: Allocate Matrix in Memory

Result: Integer pointer, pointing to first element of matrix.
Input : Total Number of Elements

```
result_ptr = (int*)calloc(number_of_elements, sizeof(int))
return result_ptr
```

A.2 getElementLocation2D

Algorithm 2: Get Location of Element in 2D matrix when represented as 1D Array

Result: Integer containing position of element in 1D matrix
Input : 1D Array containing indexes
Input : Dimension N

```
row = index[0]
column = index[1]
return (row * N) + column
```

A.3 retrieveElement2D

Algorithm 3: Retrieve Element from 2D matrix

Result: Integer value of the element in the 2D matrix co-ordinates.
Input : Pointer to matrix, 1D Array containing indexes
Input : Dimension N

```
return *(matrix_ptr + getElementLocation2D(indexes, N))
```

A.4 rank2TensorMult

Algorithm 4: 2D Matrix Multiplication Algorithm

Result: Integer pointer, pointing to first element of product matrix.
Input : Pointer to matrix A, Pointer to matrix B
Input : Matrix dimensions (N), Total number of elements in matrix

```
result_ptr = allocateMatrix(number_of_elements)
total_multiply_operations = number_of_elements · N
for m=0 to total_multiply_operations-1 do
    i = m ÷ number_of_elements
    j = m mod N
    k = (m ÷ N) mod N
    indexA = [i, k]
    indexB = [k, j]
    elementA = retrieveElement2D(matrixA, indexA, N)
    elementB = retrieveElement2D(matrixB, indexB, N)
    indexC = [i, j]
    total_ptr = result_ptr + getElmentLocation2D(indexC, N)
    *total_ptr = *total_ptr + (elementA · elementB)
return result_ptr
```

A.5 rank3TensorMult

Algorithm 5: 3D Matrix Multiplication Algorithm

Result: Integer pointer, pointing to first element of product matrix.
Input : Pointer to matrix A, Pointer to matrix B
Input : Matrix dimensions (N), Total number of elements in matrix

```
1  results3D = allocateMatrix(number_of_elements) startPos = 0
2  elements2D = N*N
3  for m=0 to N do
4      results2D = rank2TensorMult(matrix A+startPos, matrix B+startPos, N, elements2D)
5      for i=0 to elements2D do
6          results3D = results2D
7          results2D++;
8          results3D++;
9      startPos += elements2D;
10     results2D = results2D - elements2D;
11 return result3D
```