ELEN4020A: Data Intensive Computing Laboratory Exercise No 1

Kopantsho Mathafa (849038)   Chizeba Maulu (900968)   25 February 2019

I. Introduction

Multidimensional arrays are common data structures in data intensive computing. Operations performed on these give rise to models in fields such as molecular biology and climate modelling. However, performing such operations requires a careful consideration of algorithm efficiency, accuracy and scalability.

This report discusses a study performed on 3 dimensional arrays, otherwise known as 3rd-order tensors. The study employs dynamically allocated N x N x N arrays modelled in the Traditional Matrix Representation (TMR) form and applies matrix addition and multiplication operations to these arrays. The elements of these arrays are randomly generated integers valued between 0 and 20. To test for algorithm scalability, the array bound N is set at 10 and then at 20. Only the multiplication algorithm will be evaluated in this paper.

II. Background

A. Design Parameters and Success Criteria

The approach employed for the 3-dimensional matrix multiplication algorithm is built upon the canonical 2-dimensional matrix multiplication algorithm. Equation 1 illustrates 2-dimensional matrix multiplication mathematically.

$$c_{ij} = \sum_{k} a_{ik} \times b_{kj} \tag{1}$$

Where: a, b and c - 2D Matrices and i, j, and k - indices used to access the matrix elements.

Modifications are made to this well-known algorithm to account for the thired dimension in 3D matrices.

The TMR form of modelling an array is used. This was used both because of its flaws and merits. It is easier and more intuitive to model an array in TMR form as arrays in C as the array data structure is C is already in TMR form, even though in memory multi-dimensional arrays are stored as arrays of arrays instead of a single multidimensional structures.

The flaws of this form is the increase in both the cost of index computation and accessing or modifying the elements as the dimension of an array increases. These drawbacks allow for a better understanding of how algorithm efficiency can address the limitations of this kind of representation of arrays.

The requirements of the 3D multiplication algorithm and its results are:

- It must be able to dynamically generate an array of integers.
- It must populate the array with randomly generated numbers between [0,20].
- Must be able to compute the multiplication of two 3-dimensional arrays.
- The program must store the result of the computation in a result matrix C with the correct array bounds in each dimensions, or as C[N][N]][N].

B. 3-Dimensional matrix multiplication algorithm

In the design of the multiplication algorithm, the two 3D matrices are modelled as a 3D cube. This cube is then divided into sections which represent the 3 dimensions. This modelled is illustrated in Figure 1. All figures can be in the appendix.

Since the multiplication is now performed in 3 dimensions, instead of adding the products of row elements with column elements, it is necessary to compute the product of row- and column-planes. These planes are visualised in Figures 2 and 3.

Implementing this extraction of row plane and column planes, proved to be challenging. For this reason a further step was taken to divide the planes into column vectors and row vectors. This could be implemented by extracting elements using their indexed positions. A visual representation of this can be found in Figures 4 and 5.

Vector multiplication of each column and row block produce a single value, this is the value at the index where the row and column block intersect. For example, if we extracted the top row block and the leftmost column block, the matrix multiplication would be result in a value stored at Block 1 in Figure 1. This process is iterated through until the 3-dimensional result array is found.

III. Critical Analysis and Future Recommendations

The algorithm meets all the criteria outlined in section II. The accuracy of the results were confirmed manually using a scientific calculator. It was found that program correctly computed the multiplication.

The program could have been implemented to utilise parallel computing to make the computational efficiency higher. Parallel computing methods would have allowed for the use of other matrix models such the Extended Karnaugh Map Represenatation (EKMR).

The program is efficient, with regards to memory management. This is due to the use of **malloc()** and **calloc()**, which reserves only the required heap memory.

Pseudo code of the different functions can be found in the Appendix.

IV. Conclusion

This report presents the 3D matrix multiplication algorithm designed, implemented and tested. The matrices were modelled in the Traditional Matrix Representation. The elements of these matrices were randomly generated to be between 0 and 20. The algorithm computed the products of two 3D matrices with bounds 10 and 20. The algorithm performed the products accurately.

Appendix

```
function CREATEARRAY($dimension$)
    for $row = 0$; $row \leftarrow dimension$ do
        for $column = 0$; $column \leftarrow dimension$ do $matrix[row][column] = rand()\%21$
        end for
    end for
end function
```

```
function RANK2TENSORADD($A, B, rank$)
    for $row = 0$; $row \leftarrow rank$ do
        for $column = 0$; $column \leftarrow rank$ do $result[row][column] = A[row][column] + B[row][column]$
        end for
    end for
end function
```

```
function RANK2TENSORMULT($A, B, rank$)
    for $row = 0$; $row \leftarrow rank$ do
        for $column = 0$; $column \leftarrow rank$ do
            for $iterator = 0$; $iterator \leftarrow rank$ do $result[row][column] += A[row][iterator] + B[iterator][column]$
            end for
        end for
    end for
end function
```

```
function PRINTFUNC($matrix, size$)
    for $row = 0$; $row \leftarrow size$ do
        for $column = 0$; $column \leftarrow size$ do $print(matrix[row][column])$
        end for
    end for
end function
```

```
function RANK3TENSORMULT($A$,$B$,$rank$)
    for $depth = 0$; $depth \leftarrow rank$ do
        for $row = 0$; $row \leftarrow rank$ do
            for $column = 0$; $column \leftarrow rank$ do
                for $iterator = 0$; $iterator \leftarrow rank$ do $result[row][column][depth] += A[row][iterator][depth] +$
$B[iterator][column][depth]$
                end for
            end for
        end for
    end for
end function
```

```
function RANK2TENSORADD($A, B, rank$)
    for $depth = 0$; $depth \leftarrow rank$ do
        for $row = 0$; $row \leftarrow rank$ do
            for $column = 0$; $column \leftarrow rank$ do
                $result[row][column][depth] = A[row][column][depth] + B[row][column][depth]$
            end for
        end for
    end for
end function
```

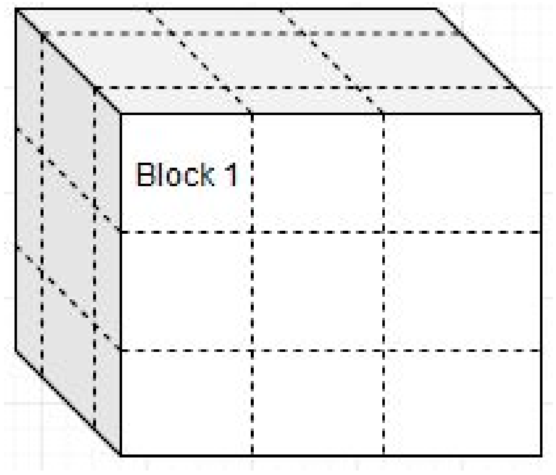

Fig. 1. Divided 3D Cube

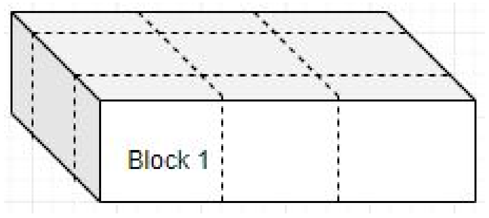

Fig. 2. Row Plane

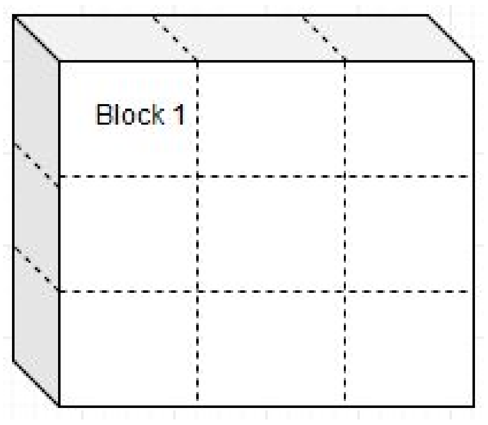

Fig. 3. Column Plane

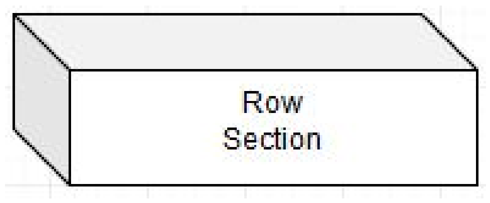

Fig. 4. Row Block

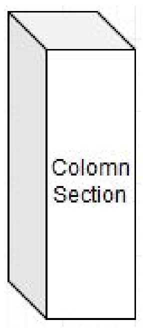

Fig. 5. Column Block