ELEN 4020 Lab 1: Tensor Arithmetic

Junaid Dawood (1094837), Xongile Nghatsane (1110680), Marissa van Wyngaardt (719804)

***Abstract*—This report discusses the basic properties of matrix multiplication as well as an attempt to formalise the process of 3D vector multiplication. These are both done with reference to a C implementation of these algorithms.**

I. Introduction

This document discusses the implementation of some simple tensor arithmetic operations in C. This extends to the description of the implemented algorithms, significant features, and pseudocode of said algorithms.

II. Tensor Structs and Helper Functions

A *struct* is implemented to represent rank 2 and rank 3 tensors. This *struct* stores the dimensions of the tensor as well as the actual N dimensional array which holds the contents of the tensor. In the case of a rank 2 tensor, the *struct* holds a row and column count as well as an *int*** representing the matrix. With rank 3 tensors, an additional height variable is present, and the *int*** is naturally replaced by a *int****.

Additional helper functions for the initialisation of the contents of the tensor and the freeing of memory dynamically allocated for their content are implemented. The helper functions, as well as the arithmetic functions, take pointers to tensor *structs* as arguments.

III. Implementation of Rank 3 Tensor Multiplication

The concept of 3D vector multiplication is not well defined. In comparison, 2D vector(matrix) multiplication is a well defined and commonly used mathematical procedure. In addition, conformity for matrix multiplication is well defined. That is, two matrices will only have a defined product if the number of columns in A is equal to the number of rows in B; in the case of $A \times B = C$.

Hence, the challenge with 3D vector multiplication is to set up a set of rules for doing so, such that the process is internally consistent. This includes the process of defining conformity for 3D vector products i.e. what dimensional equality must there be for the product of two 3D vectors to exist. One such formalization is present in [1].

The proposed definition of 3D vector multiplication is to compute a partial matrix multiplication of 'column' and 'row' matrices for each (i, j, k) in the resultant 3D vector. The 'row' matrix is defined by fixing the height value, the i coordinate in A. Similarly, if the row value, the j coordinate in B, is fixed, this will yield the 'column' matrix.

For each (i, j, k) in the resultant 3D vector, it is required that a partial multiplication of the 'row' and 'column' matrices be done. The procedure has been implemented such that each (i, j, k) will be equivalent to an element on the diagonal of the product of the 'row' and 'column' matrices. That is, for each (i, j, k) in the resultant 3D vector the row vector in A at $(i, k, :)$ will be multiplied by the column vector at $(:, j, k)$ in B.

The diagram below is provided to give a visualisation of the process decided upon.

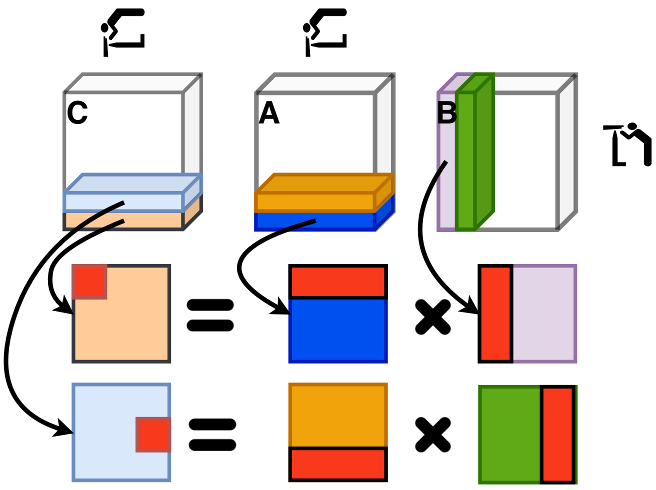

Fig. 1: 3D Vector Multiplication Visualisation.

1) Conformity

As per the definition of 3D vector given above, there must be conformity between the 'column' matrices and the 'row' matrices as per typical matrix multiplication rules. The current implementation relies on cube 3D vectors only i.e. ones which are equal in all dimensions, and returns errors when this is not the case.

IV. Additional Implementation

The concept of using a one dimensional array for the purpose of the general representation of an N dimensional array was experimented with. It is known that the naive matrix multiplication algorithm possesses $O(n^3)$ complexity, although there exists some sub-cubic implementations. For example, the Coppersmith-Winograd algorithm, the fastest algorithm currently known, achieves $On^{2.3728639}$ complexity [2].

Whilst the flattening of N dimensional arrays does not change the asymptotic complexity, it does offer potentially easier parallelisation of the matrix arithmetic procedures. At the very least, this collapses the process of matrix addition to a single *for* loop.

V. Conclusion

A discussion of matrix multiplication methodologies has been presented, alongside a formalisation of the process of 3D vector multiplication. The drawbacks of the naive implementation, as well as more efficient alternatives have also been discussed. Some thought was also given on the process of parallelising the algorithm(s), as indicated.

References

[1] A. Solo, "Multidimensional matrix mathematics: Multidimensional matrix equality, addition, subtraction, and multiplication, part 2 of 6," in *Proceedings of the world congress on engineering*, vol. 3, 2010.

[2] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, Mar. 1990. DOI: 10.1016/s0747-7171(08)80013-2. [Online]. Available: https://doi.org/10.1016/s0747-7171(08)80013-2.

Pseudocode

Algorithm 1 Rank 2 Tensor Addition

```
1: function RANK2TENSORADD(rank2Tensor t1, rank2Tensor t2)
2:     t1 ← first rank 2 tensor
3:     t2 ← second rank 2 tensor
4:     if $t1dimensions = t2dimensions$ then return ErrorResult
5:     for i=0..t1.rows do
6:         for j=0..t1.columns do
      Result[i][j]=t1[i][j]+t2[i][j]
    return Result
```

Algorithm 2 Rank 2 Tensor Multiplication

```
1: function RANK2TENSORMULT(rank2Tensor t1, rank2Tensor t2)
2:     t1 ← first rank 2 tensor
3:     t2 ← second rank 2 tensor
4:     if $t1.columns \neq t2.rows$ then return ErrorResult
       Result.rows=t1.rows Results.columns=t2.columns
5:     for i=0..t1.rows do
6:         for j=0..t2.columns do Sum = 0
7:             for  do sum += t1[i][k]*t2[k][j]
               Result[i][j]=sum
       return Result
```

Algorithm 3 Rank 3 Tensor Addition

```
1: function RANK3TENSORADD(rank3Tensor t1, rank3Tensor t2)
2:     t1 ← first rank 3 tensor
3:     t2 ← second rank 3 tensor
4:     if $t1dimensions = t2dimensions$ then return ErrorResult
5:     for i=0..t1.height do
6:         for j=0..t1.rows do
7:             for k=0..t1.columns do
      Result[i][j][k]=t1[i][j][k]+t2[i][j][k]
    return Result
```

Algorithm 4 Rank 3 Tensor Multiplication

```
1: function RANK3TENSORMULT(rank3Tensor t1, rank3Tensor t2)
2:     t1 ← first rank 3 tensor
3:     t2 ← second rank 3 tensor
4:     if $t1.columns \neq t2.rows$ then return ErrorResult
       Result.rows=t1.rows Result.columns=t2.columns
5:     for i=0..Result.height do
6:         for j=0..Result.rows do
7:             for k=0..Result.columns do Sum = 0
8:                 for l=0..t1.columns do sum += t1[i][k][l]*t2[height-1-l][j][k]
                   Result[i][j][k]=sum
       return Result
```