**Successful CUDA Fortran Matrix Multiplication with Explanation**

**1. Problem Statement**

The issue of rephrasing a simple matrix multiplication problem in terms of GPU accelerated CUDA programming is addressed in this article. The formula for matrix multiplication is described in Equation 1 below:

where C is the result of multiplication between matrix *a* and matrix *b.* The scalar *k* is the number of columns in matrix *a*. The same scalar *k* is also the number of rows in matrix *b*. Meanwhile, the scalar *i* is the number of rows of matrix *a* and matrix *c.* Meanwhile, the scalar *j* is the number of columns in matrix *c* and matrix *b*.

Analysis of Equation 1 reveals that matrix multiplication of *a* and *b* requires *i* x *j* x *k* operations. This is will take slow computation time for multiplication of very large matrices. The next Section describes a method to reduce computation time.

**2. Solution**

The operation defined in Equation 1 can potentially be done in parallel because every element of C(i,j) does not rely on its neighbours. In this document, we use CUDA to perform parallel operation. Using CUDA reduced the number of operations from *i* x *j* x *k* to just *i* x *j.*

To explore the CUDA solution, let us define a toy problem. Define matrix *a* and matrix *b* per Equation 2:

Multiplication of matrix *a* and matrix *b* yields matrix *c* per Equation (3):

The FORTRAN programming language stores 2-dimensional (2D) matrices *a, b,* and *c* by column major. It means that, we can reimagine *a, b,* and *c* as below:

The FORTRAN programming language stores the matrix *a* in computer memory by COLUMN major order per **Figure 1** below:

A red arrows and numbers

Description automatically generated

**Figure 1**. The way Fortran sees and organizes the matrix

CUDA Fortran performs individual independent calculations which are indexed. The index of CUDA Fortran is organized into grids, blocks and threads. Grids have three dimensions, x, y and z. Each dimension can have a certain number of blocks. Each block have a certain number of threads. The number of blocks and number of threads per block are user-defined. The concept of blocks and threads are further defined in the following example:

**Table 1**: Organization of CUDA blocks, threads, and index in the x-dimension.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| blockIdx%x | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 |
| threadIdx%x | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| IndexX | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

where IndexX = (blockIdx%x – 1) \* blockDim%x + threadIdx%x. In this expression, blockDim%x denotes the number of threads per block. In **Table 1,** we set the number of blocks to four (4) and the number of threads per block to four (4). In applying the CUDA configuration in **Table 1**, we express how CUDA sees matrices a, b and c:

(4)

where each number of Equation 4 represents the CUDA indexes assigned to each element of matrices *a, b*, and *c.* Note that they are assigned in COLUMN MAJOR format. This is because, FORTRAN is a COLUMN MAJOR language.

Next, we express the matrix multiplication of matrix *a* and matrix *b* in terms of CUDA indices:

The recurring theme is that, repetitions of CUDA indices (1, 5, 9, 13), (2,6,10,14), (3,7,11,15) and (4,8,12,16) happens. Another repetition of CUDA indices (1,2,3,4), (5,6,7,8), (9,10,11,12) and (13,14,15,16) happens. You can imagine this as two axes. For example, axis y is denoted by (1, 5, 9, 13), (2,6,10,14), (3,7,11,15) and (4,8,12,16). Then, axis x is denoted by (1,2,3,4), (5,6,7,8), (9,10,11,12) and (13,14,15,16).

To implement these axes, we need to map the CUDA indexes. Let’s imagine, blockIdx%x = ( 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 ). So, to get something resembling ( 1, 5, 9, 13 ), we just apply this expression:

Yes, there are duplicates, but these duplicates will not affect the final results. The duplicates have no effect on the final result, because it references to the same matrix element. For the sake of brevity, we summarize each blockIdx operation as follows:

BlockIdx is (1,2,3,4)

*blockIdx%x \* 4 - (4-k) where k varies from 1 to 4*

(1,2,3,4)\*4 – (4 – 1) = (1,5,9,13)

(1,2,3,4)\*4 – (4 – 2) = (2,6,10,14)

(1,2,3,4)\*4 – (4 – 3) = (3,7,11,15)

(1,2,3,4)\*4 – (4 – 4) = (4,8,12,16)

*blockIdx%y + (k-1) \* 4 where k varies from 1 to 4*

(1,2,3,4) – (1 – 1)\*4 = (1,2,3,4)

(1,2,3,4) – (2 – 1) \*4 = (5,6,7,8)

(1,2,3,4) – (3 – 1) \*4 = (9,10,11,12)

(1,2,3,4) – (4 – 1) \*4 = (13,14,15,16)

The FORTRAN code snippet is:

do l = 1,4

do k = 1, 4

c( blockIdx%y + (k-1) \* 4 ) = a( blockIdx%x \* 4 – (4 – k ) ) \* &

b( blockIdx%y + (k – 1) \* 4 ) + &

c( blockIdx%y + (k-1) \* 4 )

enddo

enddo

where the ampersand (&) denotes continuation line.

In summary, I explained the method of reformulating matrix multiplication into a CUDA programming problem. The code which was successful, is contained in the Github link:

In my next post, I will extend the operation to much larger arrays.