

Piyush Prabhakar Kanadje

SID:862393475

Email Id- pkana006@ucr.edu

Project Report

MATRIX MULTIPLICATION USING MULTIPLE GPU

OVERVIEW

The use of multiple GPUs concurrently is steadily growing in popularity to overcome the memory limitations of a single device and further reduce execution times. Different GPUs can work simultaneously on the problem by dividing the problem into individual computations. The matrix multiplication calculation will be divided into independent computations that will produce other parts of the final matrix, which will eventually be merged into the final matrix.

Matrix is a rectangular array or table of numbers, symbols, or expressions, arranged in rows and columns, that represents a mathematical object or one of its properties. In this project, I am performing a matrix multiplication algorithm. I am using multiple GPUs, and a matrix multiplication algorithm taught in class. In which, I am able to use multiple kernels in which I transfer data values.

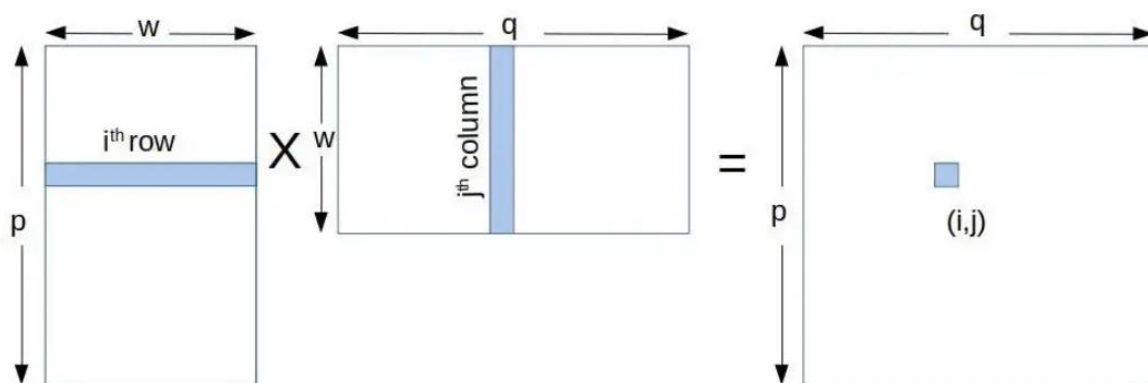
As we pass out data to the kernel, you have a set of relatively small data operations so that we would run them independently on different GPUs. Then different GPUs will execute the task in less amount of time.

Using streams, this project's code performs multi-GPU matrix multiplication with multiple kernel invocations. The program splits the computation into four separate calculations. A variable proportion of the block's size is used.

TECHNICAL DESCRIPTION

MATRIX MULTIPLICATION ALGORITHM EXPLANATION

Let's say we want to multiply matrix A with matrix B to compute matrix C. Assume A is a $p \times w$ matrix and B is a $w \times q$ matrix so that C will be the $p \times q$ matrix. Matrix multiplication is simple. Multiplying the i^{th} row of A with the j^{th} column of B will give us (i,j) the element in C. So, an individual element in C will be a vector-vector multiplication. In following figure we are able to get the basic idea of matrix multiplication: -



We are multiplying 2-D matrices. It only makes sense to arrange the thread blocks and grid in 2-D. Most modern GPUs can support up to 1024 threads per thread block. Thus, we can use a 32×32 2-D thread block (assuming `BLOCK_SIZE` x `BLOCK_SIZE` from here).

To map data to a line, we use the following mapping scheme.

```
int row = blockIdx.y*width+threadIdx.y;
int col = blockIdx.x*width+threadIdx.x;
```

In above expression we are calculating the row and column index of the element.

We know that a grid is made-up of blocks and that the blocks are made up of threads. All threads in the same block have the same block index.

To ensure that the extra threads do not do any work, we use the following 'if' condition –

```

if(row < width && col < width) {
}

```

We write the above loop in kernel and ensure that the thread do not do extra work.

```

if(row<width && col < width) {
    float product_val = 0
    for(int k=0;k<width;k++) {
        product_val += d_M[row*width+k]*d_N[k*width+col];
    }
    d_p[row*width+col] = product_val;
}

```

Then using for loop we calculate the value of the product_val(i,j) and then store that value in the output matrix.

MULTI GPU MULTIPLICATION IMPLEMENTATION

In this section I will explain the how multi-GPU multiplication is done.

First thing first we check the GPUs are available are not if available how many of them are available.

```

/*****
Multi GPU Multiplication is done by splitting the comutation into 4 parts as follows:
A * B = C
| A1 |   |   |   |   C1 | C2
----- * | B1 | B2 | = -----
| A2 |   |   |   |   C3 | C4

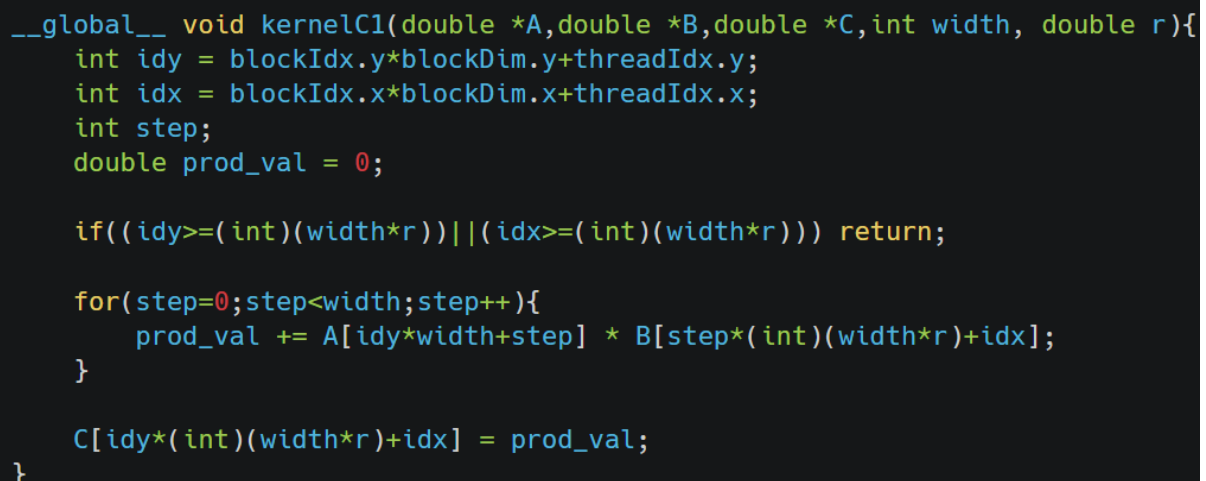
A1 * B1 = C1
A1 * B2 = c2
A2 * B1 = C3
A2 * B2 = C4
These 4 individual computations take place simultaneously on different GPUs.
*****/

```

As we can see in above figure for the multiplication of $A * B = C$. We first Split A into 2 different parts and after that we also split B in two different parts. After that we get 4 different parts to compute the multiplication.

To handle these 4 individual computations, I have defined 4 different kernels which are identical as each other. In which I have use logic same as simple matrix multiplication as shown above. After all the separate computations have completed on the different devices, then we merge the intermediate results to create the final result.

With CUDA, we implemented this functionality by using `cudaSetDevice(int device)`, which specifies the device on which the active thread executes the device code. CUDA Streams are also used instead of the default to invoke the kernels simultaneously on many streams. In that way, we can achieve concurrency.



```
__global__ void kernelC1(double *A, double *B, double *C, int width, double r){
    int idy = blockIdx.y*blockDim.y+threadIdx.y;
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    int step;
    double prod_val = 0;

    if((idy >= (int)(width*r)) || (idx >= (int)(width*r))) return;

    for(step=0; step<width; step++){
        prod_val += A[idy*width+step] * B[step*(int)(width*r)+idx];
    }

    C[idy*(int)(width*r)+idx] = prod_val;
}
```

Above figure indicates the Kernel which is used for the computation of first streams of data in which we are storing value in variable C.

LIBRARIES USED

`<stdint.h>`, `<stdlib.h>`, and `<stdio.h>` standard C libraries were used that provide a set of functions and data types for working with integers, memory allocation, and input/output operations.

I also have used libraries `<time.h>` which is C date and time functions are a group of functions in the standard library of the C programming language

implementing date and time manipulation operations. Also, `<unistd.h>` is the name of the header file that provides access to the POSIX operating system API.

STATUS OF THE PROJECT

My project uses a matrix multiplication algorithm in which we pass the data and then split it into four different parts for the computation. To handle four different computations, I have written code for the four different kernels in which I am able to get the results I wanted. The Status of the project is that I am able to see the that algorithm is able to compute the matrix multiplication using 4 different GPU. The Feature for scaling to the multiple GPU works well.

CHALLENGES OF THE PROJECT

While implementing the project I have referred the existing documentation and implementation of the same multi-GPU matrix multiplication but in every implementation the matrix is only able to handle $1000 * 1000$ matrix and no other matrixes. Also, the implementation was unable to perform matrix multiplication on odd value size of matrix. So, I have implemented the code in which I am able to perform matrix multiplication of odd value size matrix and matrix size other than $1000 * 1000$.

Evaluation And Result On Next Page →

EVALUATION AND RESULTS

Multiple GPU Implementation 1000 * 1000 Matrix Multiplication

```
Available Number of GPU: 4

Time For Setting Up 0.246193 s
  A: 1000 x 1000, 1000000 elements
  B: 1000 x 1000, 1000000 elements
  C: 1000 x 1000, 1000000 elements

Width Of Block 1: 500
Width Of Block 2: 500

Allocating the parts of data to Kernel 1...0.004806 s
Allocating the parts of data to Kernel 2...0.105934 s
Allocating the parts of data to Kernel 3...0.106269 s
Allocating the parts of data to Kernel 4...0.105665 s

Copying data from host to device 1... 0.001758 s
Execution Of Kernel1... 0.000052 s

Copying data from host to device 2... 0.000716 s
Execution Of Kernel 2... 0.000012 s

Copying data from host to device 3... 0.001209 s
Execution Of Kernel 3... 0.000009 s

Copying data from host to device 4... 0.000013 s
Execution Of Kernel 4... 0.000006 s

Copying data from devices to host... 0.006052 s
Building The Final Matrix from Block
Building final matrix from blocks... 0.001765 s

TEST PASSED 1000000

Verifying results... 1.729557 s

Freeing Host and Device Memory... 0.011577 s
```

As we can see in the above snapshot First Line is telling us the Available no of GPU- 4. And then we are Able to send data to different kernels and able to compute final Matrix in 0.001765s.

Single GPU implantation of 1000 * 1000 matrix execution:

```
Setting up the problem... 0.028017 s
  A: 1000 x 1000
  B: 1000 x 1000
  C: 1000 x 1000
Allocating device variables... 0.201492 s
Copying data from host to device... 0.001939 s
Launching kernel... 0.004656 s
Copying data from device to host... 0.003633 s
Verifying results... TEST PASSED 1000000
```

As we can see the total execution of the single 1000 * 1000 matrix is 0.003633s.

Sr No.	MATRIX SIZE	MULTI GPU TIME	SINGLE GPU TIME
1	1000 * 1000	0.001765s	0.003633s
2	3000 * 3000	0.002856s	0.014088s
3	5000 * 5000	0.003493s	0.041869s

As we can infer from the table that for the multiple GPU execution to perform matrix multiplication is less than the execution of single GPU.

CONCLUSION

Using multiple GPU for the matrix multiplication executes the algorithm in less time and we should use multiple GPU rather than using single GPU. In this Cuda Streams Provides us a way to execute multiple independent tasks on multiple GPU concurrently. This provides us the faster implementation of algorithm.

DOCUMENTATION ON STEPS TO RUN CODE

To create and run the Matrix Multiplication on Multiple GPU, please follow below steps:

1. Clone the repository containing the source code by running the command `git clone`. This will download the code to your local machine.
2. Navigate to the directory containing the source code and run the `make` command to compile the code. This will create the compiled files that are needed to run the program.
3. To run the program, use the command `./multipleGpu`. By default, this will run the program with 1000 * 1000 matrix, and will verify the result.