Matrix Convolution using Parallel Programming

Anirud Pande[1], Rohit Chandna[2]

[1, 2] Department of Computer Science & Engineering, Manipal Institute of Technology
Manipal, India

Abstract: ***The convolution theorem is used to multiply matrices of two different sizes i.e. matrices in which the number of rows in the first matrix is not equal to the number of columns in the second matrix. In this study, the multiplication of 3*3 and 4*4 matrices was done using MPI.A 3*3 matrix was taken as a filter which was multiplied with different matrices of sizes as big as 1000*1000 and was implemented using OpenCL. Each element of the resultant matrix was calculated both parallelly and sequentially and their performance and efficiency were compared on the basis of execution time.***

Keywords: Convolution filter, Bitmap, OpenCL, Kernel

1. Introduction

The process to calculate the multiplication of two matrices sequentially is lengthy. OpenCL & MPI allow us to calculate the product in an efficient manner. The time taken to execute the program is calculated both sequentially and parallelly and a graph of time versus size of the matrix is plotted. This graph depicts which method of execution is better. Mathematical representation of convolution, (convolution theorem), which gives the inverse Laplace transform of a product of two transformed functions, is discussed. Convolution filter, an application of convolution is also discussed.

The paper examines the terminologies, benefits of open computing language (OpenCL), sequential and parallel paths of execution. It also discusses various OpenCL architecture models and procedures to calculate each element. The implementation of matrix multiplication using OpenCL, message passing interface (MPI), syntax of various commands used in MPI, and commands used for initialization are discussed.

2. Theory

2.1 Mathematical Representation

For any x, y ϵ C(x* y)$\leftrightarrow$ X.Y[1] (1)
Convolution is a simple mathematical operation which is used by many common image processing operators. The convolution theorem specifies that the applying convolution is the same as a per-frequency multiplication in the frequency domain i.e. if the basis for both the convolution kernel and the image were to be changed to one that consists of simple sine and cosine functions by applying a discrete Fourier transform then we can take each of these components, multiply them and get the same result. This means that Fourier transform of the convolution kernel can be taken and the dampened frequencies can be seen (those having an amplitude<1), strengthen (>1) or leave unchanged (=1).A maximum amplitude value of one indicates that each of the different frequencies are independently attenuated, i.e. the frequency components in an image can be filtered out.

The convolution of 2 functions f (t) and g(t) is denoted by (f * g) (t) [2]
Convolution theorem gives the inverse Laplace transform of a product of two transformed functions:

$L^{-1}\{F(s)\ G(s)\} = (f * g)(t)$ (2)

Let f (t) and g (t) be two functions of t. The convolution of f (t) and g (t) is also a function of t, denoted by (f * g) (t).

And is defined by the relation

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t-x)\ g(x)\ dx \quad (3)$$

However, if f and g are both causal functions then f (t) and g (t) are written as f (t)u (t) and g (t)u (t) respectively, so that

$$(f * g) = \int_{-\infty}^{\infty} f(t-x)\ u(t-x)\ g(x)\ u(x)\ dx$$
$$= \int_{-\infty}^{\infty} f(t-x)\ g(x)\ dx$$

Because of the properties of the step functions (u(t – x) = 0 if x>t and u(x) = 0 if x<0).

2.2 Convolution Filter

Convolution Filter is used to combine pixel data in a bitmap with data from neighboring pixels to produce a given result. A wide array of effects can be produced on a bitmap by having control at the pixel level. These include things like blurring, beveling, embossing, sharpening, and more. All are possible using Convolution Filter. Convolution Filter's matrix does not have a set number of rows and columns. The number of rows and columns depend on the type and strength of the effect you are trying to achieve. Thus, Convolution Filter looks at each and every pixel in a source bitmap and as it does this, it uses the center value in the matrix as the value of the current pixel being manipulated. For example, in a 5 x 5 matrix, the center value is at (2, 2). The values from the matrix are multiplied to the surrounding pixels and the resulting values for all pixels are added to get the value for the resulting center pixel. The formula used on a 3 x 3 matrix convolution is:

dst (x, y) = ((src (x-1,y-1)*a0 + src (x,y-1)*a1 (4)
src(x, y+1) * a7 + src (x+1,y+1) * a8)/divisor) + bias (5)

Convolution Filter with a 3 x 3 matrix takes the pixel (x–1, y–1) for the pixel located at (x, y) and multiplies it by the value in the matrix located at (0, 0), and then adds the pixel (x, y–1) multiplied by the value in the matrix at (0, 1), and

so on until all of the matrix values have been multiplied by the corresponding pixel value. (This is done for each color channel.) Finally, it finds out the sum, divides it by the value of divisor, and adds the value of bias. Obviously the larger your matrix, the longer this process takes. In image processing, many operators are based on applying some function to the pixels within a local window i.e. for finding the value of an output pixel, a window is centered at that location, and only the pixels falling within this window are used when calculating the value of that output pixel.

When applying the convolution operator, the function we apply is merely a weighted average of the within-window pixels. The function can be defined by providing a 5x5 weight- matrix if the window size is 5×5 pixels. So, at each pixel in the image, we place this 5x5 matrix and perform element-wise multiplications before summing up. This sum is deemed the output value at that location. If we let x be the image we want to filter, y the corresponding output image, and let h be the convolution filter matrix, we have:

$$Y=h*x \qquad (6)$$

The convolution operator is linear, i.e. we get the same result if we perform the convolution on two separate images and sum their results as if we were to sum the two images before we apply the convolution.

$$y[1,1] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j].h[1-i,1-j]$$

$$= x[0,0].h[1,1] + x[1,0].h[0,1] + x[2,0].h[-1,1] + x[0,1].h[1,0] + x[1,1].h[0,0] + x[2,1].h[-1,0] + x[0,2].h[1,-1] + x[1,2].h[0,-1] + x[2,2].h[-1,-1]$$

Hence, in convolution 2D with M×N kernel, it requires M×N multiplications.

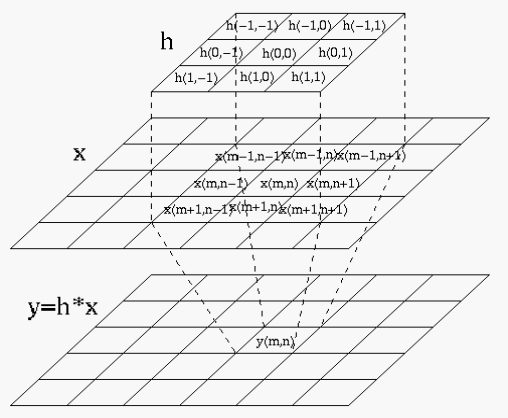

Figure 1: Multiplication of matrix procedure

Let us assume that the size of filter matrix h, is 3x3, and its values are a, b, c, d, e, f, g, h, i

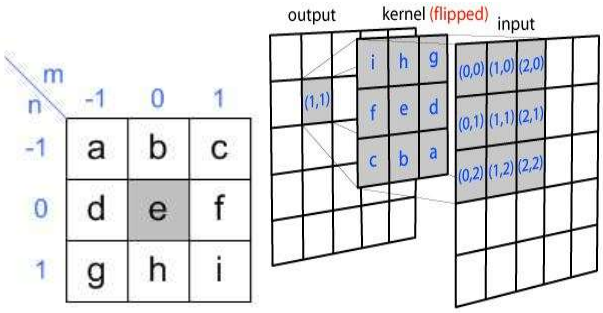

Figure 2: Depiction of finding an element of output matrix

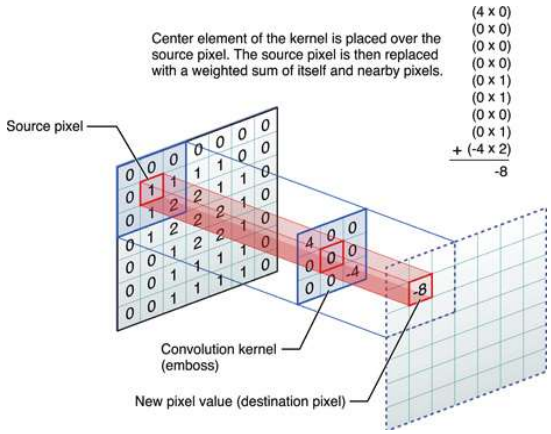

Figure 3: Relation between source and destination pixel

In image filtering you should have a 2D filter matrix and the 2D image. Then you can take the sum of products for every pixel of the image. Each product is the color value of the current pixel or a neighbor of it, with the corresponding value of the filter matrix. The center of the filter matrix has to be multiplied with the current pixel and the other elements of the filter matrix with corresponding neighbor pixels. This operation where you take the sum of products of two 2D functions, where you let one of the two functions move over every other element of the other function is called Convolution or Correlation. The difference between Convolution and Correlation is that for Convolution you have to mirror the filter matrix. But usually it's symmetrical anyway so there is no difference.

The 2D convolution operation requires a 4-double loop, so it isn't extremely fast, unless you can use small filters. Here we will usually be using a 3*3 or 5*5 filters.

There are a few rules about the filter:

a) Size: Its size has to be uneven, so that it has a center element, for example 3x3, 5x5 and 7x7.
b) Sum equal to 1: It doesn't have to, but the sum of all elements of the filter should be 1 if you want the resulting image to have the same brightness as the original.
c) Sum greater than 1: If the sum of the elements is larger than 1, the result will be a brighter image.
d) Sum smaller than 1: If it's smaller than 1, a darker image.
e) Sum is equal to 0: If the sum is 0, the resulting image isn't necessarily completely black, but it'll be very dark.

The image has finite dimensions. So for calculating a pixel on the left side, there are no more pixels to the left of it

while these are required for the convolution. Either the value 0 can be used here, or it can be wrapped around to the other side of the image. The wrapping around is preferred because it can easily be done with modulo division.

The resulting pixel values after applying the filter can be negative or larger than 255, if that happens then the resulting values can be truncated so that values smaller than 0 are made 0 and values larger than 255 are set to 255. The absolute value can also be taken for negative values. The convolution operation becomes a multiplication instead in the Fourier Domain or Frequency Domain, so it is faster. In the Fourier Domain, much more powerful and bigger filters can be applied faster, especially if the Fast Fourier Transform is used. It is not yet feasible to use image filters for real time applications and games but they are useful in image processing. Digital audio and electronic filters work with convolution as well, but only in 1D.

2.3. OpenCL Basics

The Open Computing Language (OpenCL) is an open and royalty-free parallel computing API that enables GPUs and other coprocessors to work concurrently with the CPU, thus providing additional raw computing power.

2.3.1 Benefits of OpenCL

A primary benefit of OpenCL is considerable acceleration in parallel processing. OpenCL uses the resources available in the system by taking all computational resources, such as multi-core CPUs and GPUs, as peer computational units and correspondingly allocating different levels of memory. OpenCL also complements the existing OpenGL visualization API by sharing data structures and memory locations without any copy or conversion overhead. A secondary benefit of OpenCL is that it is cross-vendor software portable. This low-level layer draws an explicit line between hardware and the upper software layer. All the hardware implementation specifics, like drivers and runtime are made invisible to the upper-level software programmers through the use of high-level abstractions. This allows the developer to use the best hardware without having to reshuffle the upper software infrastructure. The change from proprietary programming to open standard also helps in the acceleration of general computation in a cross-vendor fashion.

2.3.2 Open CL Architecture

(a) The Platform Model

The OpenCL platform model describes a host connected to one or more OpenCL devices. The fig 4 given below shows the platform model consisting of one host with multiple processing elements. A host is any computer which consists of a CPU and a standard operating system. An OpenCL device is a collection of one or more compute units(cores) and such devices can be GPU, DSP, or a multi-core CPU. A compute unit consists of one or more processing elements. The processing elements execute instructions as SIMD (Single Instruction Multiple Data) or SPMD (Single Program, Multiple Data). SPMD instructions are executed on general purpose devices like CPUs whereas SIMD instructions need a vector processor such as a GPU or vector units in a CPU.

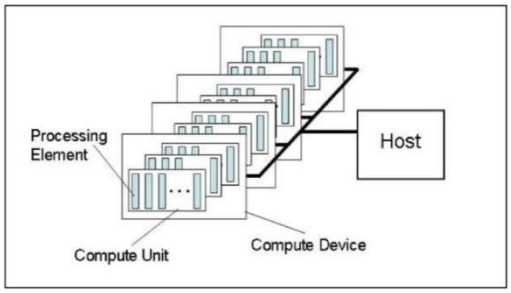

Figure 4: The Platform Model

(b) The Execution Model

The OpenCL execution model consists of two components: Kernel and Host Programs. Kernels are the basic unit of executable code that run on one or more OpenCL devices. Kernels are like a C function which can be data or task parallel. The host program defines devices context, and queues kernel execution instances using command queues. It is executed on the host system. Kernels are queued in-order, but can be executed in-order or out-of-order.

Kernel: An index space is defined when a kernel is queued for execution by the host program. Each independent element of execution in this index space is called a work-item which executes the same kernel function but on different data. The N-dimensional index space can be N=1, 2 or 3. Work-items can be grouped together into work-groups. Local index space defines the size of work-group. All work-items in the same work-group are executed together on the same device. This allows work- items to share local memory and synchronization.

Host Program: The host program sets up and manages the execution of kernels on the OpenCL device through use of context. Using OpenCL API, the host can create and manipulate the context by including the following resources:

- Devices: A set of OpenCL devices used by the host to execute kernels.
- Program Objects: Objects that implement a kernel or collection of kernel.
- Kernels: The specific OpenCL functions that execute on the OpenCL device.
- Memory Objects: A set of memory buffers common to the host and OpenCL devices.

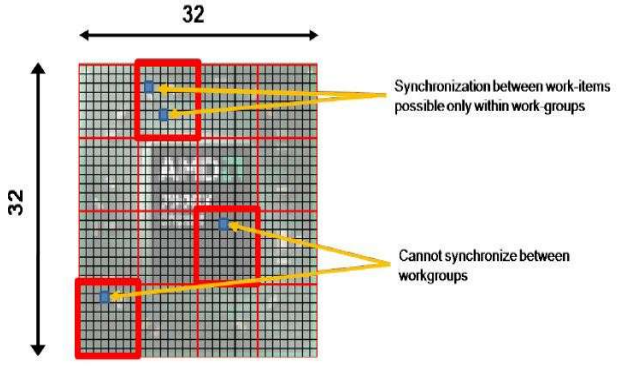

Figure 5: The Execution Model

(c) Memory Model

The OpenCL memory models define four regions of memory accessible to work-items when a kernel is executed. The figure given below shows the region of memory accessible by host and the compute device. Global Memory: It is a region in which all work-items and work-groups have read and write access on both the compute device and the host. This region of memory can be allocated only by the host during runtime.

- Constant Memory: It is a region of global memory that stays constant throughout the execution on the kernel. Work-items have only read access to this region, while the host is permitted both read and write access.
- Local memory: It is a region of memory used for data-sharing by work-items in a work-group. All work-items in the same work-group have read and write access.
- Private memory: It is a region that can be accessed by only one work-item.

In most cases, host memory and compute device memory are independent of one another. To ensure memory management between host and the compute device, this process en-queues read/write commands in the command queue.

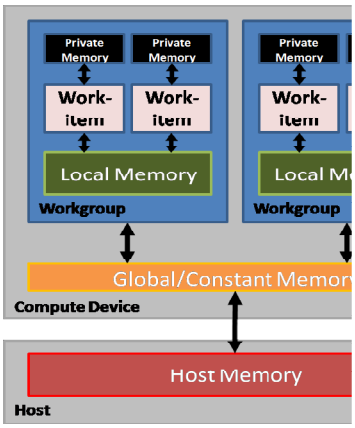

Figure 6: The Memory Model

3. Methodology

3.1 Basic Multiplication Method

'a' is an m*n matrix, with elements x_0,x_1,x_2 ….x_{n-1}& 'd' is another n*m matrix with elements y_0,y_1,y_2 ….y_{n-1}. Dot product is:

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j (7)$$

3.1.1 Sequential Way

```
for (i = 0; i<m; i ++)
{
        Y[i]=0;
/*for each element of the row and each element of x*/
for (j=0; j<n; j ++)
        Y[j] + = a[i][j] 56 * x[j];
}
```

3.1.2 Parallel Way

We divide the iteration of the outer loop among the threads. To compute y [0], process 0 need to execute the code:

```
Y[0]=0;
for(j=0;j<n;j++)
Y [0]+=a [0][j]*x[j];
```

3.2 Implementation

3.2.1 OpenCL Implementation

Convolution is the treatment of a matrix by another one called the kernel. The convolution matrix filter uses the image to be treated as the first matrix. The image is a bi-dimensional collection of pixels in rectangular coordinates. Only 3*3 matrices will be considered as they are mostly used and are enough for all the effects we want. The second matrix has a variable size. Then the matrices are converted to 1-D array and the two array of numbers, which are generally of different sizes but of the same dimension are multiplied. It produces a third array of numbers of the same dimensionality. This can be used in image processing to implement operators whose output values are simple linear combinations of certain pixel values.

3.2.2 Message Passing Interface (MPI)

In this code we multiply an n*n matrix with an m*m matrix using convolution theorem. m+2 processes are there that compute each elements of final matrix in parallel.

(1) Initialization

MPI_Init (&argc,&argv):This function initializes the message passing interface by passing arguments on command line. They are 0 by default. MPI_Comm_size(MPI_COMM_WORLD,&totalnodes):This function gives total number of processes. First parameter is communicator. Second parameter contains the total number of processes.
MPI_Comm_rank(MPI_COMM_WORLD & mynode):This function gives the rank of the process, which is 1 less than total number of processes.

(2) Steps

Two matrices are input in process 0 of rank 0. Then the size of padded matrix c is sent to process 1 using MPI_Send method. This initializes all elements of c to 0, which is (m+2) * (m+2) matrix. This is received by the 0^{th} process by MPI_Recv method, which adds the elements of matrix b to matrix c. Hence only outer elements of c are 0. Initialize the m*m size output matrix d using same procedure as given above. Then find out each element of the resultant matrix. First send the size of the output matrix & padded matrix (using MPI_Send) to all the processes. Different processes calculate the elements of output matrix and return the result to process 0, using MPI_Recv.
That is how output will be received from m processes concurrently.

(3) Commands Syntax

(a)MPI_Send()

It performs a blocking send.
Synopsis:intMPI_Send(void *buf, int count, MPI_Datatypedatatype, intdest, int tag,MPI_Commcomm)
Input Parameters:
buf: initial address of send buffer (choice)

count: number of elements in send buffer (nonnegative integer)
datatype: datatype of each send buffer element (handle)
dest: rank of destination (integer)
tag:message tag (integer)
comm.: communicator (handle)

(b)MPI_Recv()
It is a blocking receive for a message.
Synopsis: intMPI_Recv(void *buf, int count, MPI_Datatypedatatype, int source, inttag,MPI_Comm comm, MPI_Status *status)
Output Parameters:
buf: initial address of receive buffer (choice)
status: status object (Status)
Input Parameters
count: maximum number of elements in receive buffer (integer)
datatype: datatype of each receive buffer element (handle)
source:rank of source (integer)
tag: message tag (integer)
comm: communicator (handle)

(c) MPI_Finalize()
This function tests if work of a process is finished.

4. Analysis

4.1 Sequential Execution

Table 1: Sequential execution of OpenCL code

Matrix Dimension	*Kernel Execution*	*Program Execution*
4*4	28.532	0.331
	27.626	0.305
	24.908	0.339
10*10	36.231	0.892
	38.043	0.636
	36.684	0.589
50*50	554.790	1.526
	367.293	1.446
	432.056	1.576
100*100	1389.464	2.192
	1803.404	3.200
	1526.689	3.213
500*500	31083.139	55.309
	32685.008	53.880
	45005.403	53.945
1000*1000	127867.354	227.709
	136517.536	174.916
	126118.296	198.392

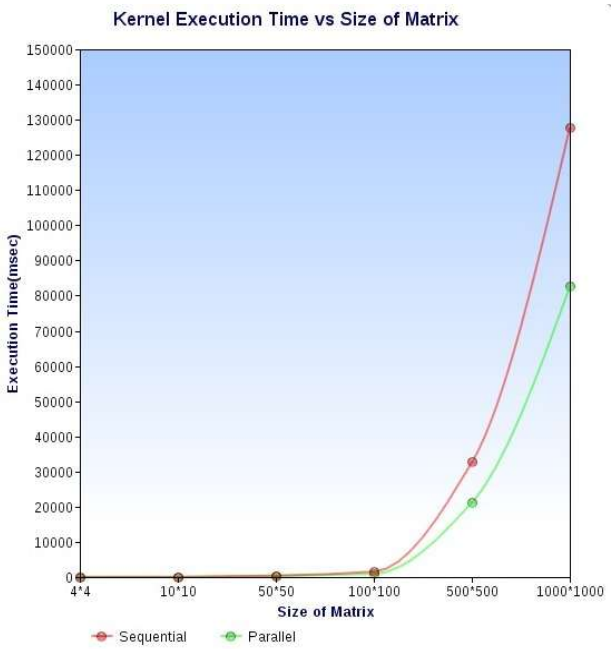

Figure 7: A simple line graph showing relation between Kernel execution time and size of matrix in sequential and parallel execution

4.2 Parallel Execution

Table 2: Parallel execution of OpenCL code

Matrix	Kernel Execution	Program Execution
4*4	77.897	0.381
	36.684	0.301
	66.575	0.368
10*10	50.724	0.670
	44.836	0.666
	47.100	0.367
50*50	294.831	0.835
	224.633	1.016
	253.618	1.084
100*100	885.851	1.858
	911.665	2.551
	893.550	3.102
500*500	37559.453	50.396
	21312.962	51.545
	17800.806	51.337
1000*1000	82606.974	165.967
	83834.304	127.658
	80695.361	126.406

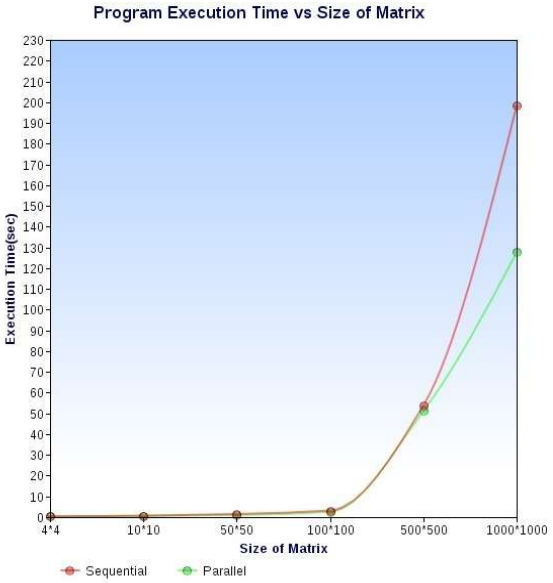

Figure 8: A simple line graph showing relation between Program execution time and size of matrix in sequential and parallel execution**.**

Above estimates show that for matrices of small sizes such as 4*4 and 10*10 sequential execution takes less time, whereas for large size matrices such as 500*500 and 1000*1000, parallel execution takes less time than compare to sequential execution.

5. Conclusion

Executing matrix multiplication for matrices of small sizes sequentially rather than parallelly takes less time. With increasing size of the matrices, it takes less time to execute the code parallelly rather than sequentially, therefore parallel execution of code is more efficient for data of large sizes.

References

[1] Convolution Theorem original form [Online]. Available:https://ccrma.stanford.edu/~jos/st/Convolution_Theorem.html
[2] Engineering Mathematics: Open Learning Unit Level, The Laplace Transform.
[3] Convolution Matrix, Generic filters [Online]. Available:http://docs.gimp.org/en/plug-in-convmatrix.html
[4] Convolution Based Filters [Online].Available: http://www.imageprocessingbasics.com/image-convolution-filters/
[5] Image Filtering [Online]. Available: http://lodev.org/cgtutor/filtering.html
[6] Image Convolution with CUDA by Victor Podlozhnyuk.
[7] Surrounding Theorem: Developing Parallel Programs for Matrix-Convolutions by KentoEmoto,
[8] KiminoriMatsuzaki, Zhenjiang Hu, and Masato Takeichi, Department of Mathematical Informatics, University of Tokyo.
[9] MPI Details [Online]. Available:www.khronos.org

Author Profile

Anirud Pande is a 4th year B Tech Computer Science Engineering student at Manipal Institute of Technology, India. He is currently doing internship from National Informatics Center, Delhi.

Rohit Chandna is a 4th year B Tech Computer Science Engineering student at Manipal Institute of Technology, India. He is currently doing internship from Indiareads, Noida.