World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:9, No:7, 2015

Detecting the Edge of Multiple Images in Parallel

Prakash K. Aithal, U. Dinesh Acharya, Rajesh Gopakumar

Abstract**—**Edge is variation of brightness in an image. Edge detection is useful in many application areas such as finding forests, rivers from a satellite image, detecting broken bone in a medical image etc. The paper discusses about finding edge of multiple aerial images in parallel. The proposed work tested on 38 images 37 colored and one monochrome image. The time taken to process N images in parallel is equivalent to time taken to process 1 image in sequential. Message Passing Interface (MPI) and Open Computing Language (OpenCL) is used to achieve task and pixel level parallelism respectively.

Keywords**—** Edge detection, multicore, GPU, openCL, MPI.

Open Science Index, Computer and Information Engineering Vol:9, No:7, 2015 publications.waset.org/10001497.pdf

I. Introduction

A. Edge Detection

EDGE is sharp variation of brightness in an image. Edge detection is a preprocessing step in an image analysis application. There are many sequential version of edge detection algorithm. Edge can be detected in OpenCL. OpenCL provides pixel level parallelism by creating as many threads as number of pixels in an image. Thus OpenCL provides facility for parallelization of edge detection of single image. Proposed work uses MPI for creating as many processes as number of images. Each MPI process calls OpenCL kernel code achieving image level parallelism.

B. MPI

MPI is a parallel library which is used with language C. It is suitable for message passing parallel computer. In message passing model Applications are co-operating processes with local variables. Some of the most commonly used API in MPI are listed in the Table I.

TABLE I
API'S IN MPI

API NAME	PURPOSE
MPI_Init(int *argc, char ***argv);	Initialize the MPI Environment
MPI_Comm_rank(Communicator,&rank)	Returns number of tasks in the communicator
MPI_Finalize()	Finalize the MPI Environment

In MPI each process will have its local memory and there is no shared memory. MPI is a Multiple Instruction stream Multiple Data stream (MIMD) system. MPI is tightly coupled system. MPI is a popular message passing interface used for communication in parallel computers. It follows the MIMD (multiple instruction multiple data) type of parallelism. In this type of parallelism, the processors may operate at different speeds and therefore such systems are asynchronous. Thus, the synchronization of the processes in an MIMD system becomes a critical problem. However, MPI contains several functionalities in its interface that provide synchronization and communication between a set of processes. It also allows a number of processes to be created on several nodes which can operate in a parallel manner. In the project, a number of MPI processes are started so that each process can read in an image and extract its features.

There exist MPI I/O operations which can control file reads and writes so that the processes can all access a common file and read from it or write to it in a synchronous manner without overwriting each other's values. The feature vectors that have been obtained and created by each MPI process is written synchronously to a file using these MPI I/O operations for synchronization.

C. OpenCL

OpenCL is another parallel library used with C language. It is a Single Instruction stream Multiple Data stream (SIMD) system. Following are the steps to be followed for any of the OpenCL Application.

Step1. Discover and initialize the platforms: To discover and initialize the platform clGetPlatformIds(…) is called twice. At the first call number of available platforms are discovered. Enough memory is allocated to the platforms then second call to the same function initializes the platform.

Step2. Discover and initialize the devices: This step is similar to discovering and initializing platforms the clGetDeviceIds(…) function is called twice. One call is to discover the number of devices. Allocate sufficient memory to the devices and then second call initializes the devices.

Step3. Create a context: A context is the abstract container that resides on host. It manages the memory objects and keeps track of the kernels that are created for each device.

Step4. Create a command queue: One command queue is created per device.

Step5. Create device buffers or images: This step creates the memory object which OpenCL can understand. Memory objects can be either large arrays called buffers or images.

Step6. Write host data to device buffers: Devices can execute the data which are either buffers or images.

Step7. Create and compile the program

Step8. Create the kernel

Step9. Set the kernel arguments

Prakash K Aithal is with the Manipal Institute of Technology, Manipal University, Manipal, India (phone: 9535829916; e-mail: prakash@manipal.edu).

U Dinesh Acharya and Rajesh Gopakumar are with the Manipal Institute of Technology, Manipal University, Manipal, India (e-mail: dinesh.acharya@manipal.edu, rajesh.g@manipal.edu).

Open Science Index, Computer and Information Engineering Vol:9, No:7, 2015 publications.waset.org/10001497.pdf

Step10. Configure the work-item structure: Work item will be of one dimension for arrays, two dimensions for matrices and three dimension for images.
Step11. Enqueue the kernel for execution: Execute the kernel on devices.
Step12. Read the output buffer back to the host: Transfer the data from device to the host.
Step13. Release OpenCL resources: All the resources used in the OpenCL program must be released.

OpenCL is a framework for writing programs that can execute across heterogeneous platforms such as CPUs and GPUs. It provides parallel computing using task-based and data-based parallelism. Heterogeneous systems include special massively parallel accelerators that can be considered as math co-processors such as GPUs. A program for hybrid architecture can be split into a CPU code (called a host code) and a code for the GPU (called a kernel code).

OpenCL views a computing system as consisting of a number of compute devices. These compute devices may be CPUs or GPUs and are all attached to a host processor. A compute device consists of many individual processing elements and a single kernel execution can run on all or many of the processing elements in parallel.

On submission of the kernel by the host to the device, an N dimensional index space is created. Each kernel instance is created at each of the coordinates of this index space. This instance is called a kernel work item. Kernel work groups each contain an equal number of work items and each does its computations on a separate compute node. Each work item is assigned a unique global ID and a local ID that is unique to the work group it is in. Each work group is also assigned a unique work group ID. The assignment of IDs is to be able to identify each and every work item separately.

The OpenCL memory model guarantees a relaxed memory consistency between devices. This means that different work items may see a different view of the global memory as computation progresses. Synchronization is thus required to ensure data consistency within the work items in a work group. There are 4 different types of memory that a work item can access:

1) Global Memory

This memory region is device wide and changes made in this region and visible to all the work items.

2) Local Memory

Each OpenCL device has an associated local memory. This is the memory that is the closest to the OpenCL processing element. That is, if a work item modifies the local memory, the change is made visible to all the work items in the work group but cannot be seen outside the work group.

3) Constant Memory

This is the region of memory that remains constant throughout the execution time of the kernel and is initialized by the host device

4) Private Memory

In this region of memory is used to allocate all the local variables in the kernel code. Any modifications done to this memory are not visible to other work items.

The detailed description of steps can be found in [1].

II. Literature Survey

Edges characterize boundaries and are therefore a problem of fundamental importance in image processing. Image Edge detection significantly reduces the amount of data and filters out useless information, while preserving the important structural properties in an image. Since edge detection is in the forefront of image processing for object detection, it is crucial to have a good understanding of edge detection algorithms [2]. Mohammed Baydoun et al. have proposed parallel approach for range filtering, sharpening and histogram equalization [3]. Prakash K. Aithal et al. have proposed parallel edge detection of coloured images using MPI by slicing the image horizontally and feeding it to multiple processes in parallel. The edge detected image is then combined using image reconstruction algorithm [4]. Varun Sanduja et al. have proposed edge detection based on FPGA which uses Sobel filter. Sobel filter needs to be applied in x and y direction [5]. Gradient based edge detection, Laplacian based edge detection using different filters can be found in [2].

The remaining paper is organized as follows Section III discusses sequential edge detection approach. Section IV discusses edge detection in parallel using MPI. Section V explains edge detection using OpenCL. Section VI elaborates edge detection of multiple images in parallel with the help of combination of OpenCL and MPI. Section VII presents the results.

III. Sequential Edge Detection Using Laplacian of Gaussian Filter

In sequential edge detection algorithm, convolution matrix is found by applying Laplacian of Gaussian filter on image matrix. If Sobel or Prewitt has two filters, one in X-direction and other in Y-direction. Laplacian of Gaussian filter has the advantage that only one filter is sufficient thus reducing the cost of computation by the factor of 2. A 5x5 Laplacian of Gaussian filter is depicted in Fig. 1.

$$\begin{matrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{matrix}$$

Fig. 1 5 x 5 Laplacian of Gaussian filter

Sequential edge detection algorithm operates upon one complete image using convolution matrix. The given image is multiplied with the filter to generate convolution matrix. Different filters can be applied to sharpen the image, blur the image, emboss and detect the edge [4].

Open Science Index, Computer and Information Engineering Vol:9, No:7, 2015 publications.waset.org/10001497.pdf

IV. Edge Detection through MPI

An Image is cut horizontally and fed to the different processes. Each process finds the edge of sliced image. Image is reconstructed by combining edges detected by each of the processes. Edge detection through MPI can be done as follows first slice the image horizontally and feed each slice to a process in parallel. Apply filter and get the convolution matrix. Combine the convolution matrix or reconstruct the image using the algorithm given in [4].

V. Edge Detection through OPENCL

Edge of an image is calculated by pixel level processing of an image. Kernel code runs on GPU devices. The steps to identify edge are listed below.

1. Read the image
2. Convert 24 bit RGB to 32 bit RGBA format.
3. Create space for bufferSourceImage using CreateImage2D API (Note: Width field will be ImageWidth and Height field will be ImageHeight).
4. Create space for bufferOutputImage using CreateImage2D API. (Note: Width field will be ImageWidth-filterWidth+1 and Height field will be ImageHeight-filterHeight+1).
5. Create Space for bufferFilter using clCreateBuffer API.
6. Write the image onto bufferSourceImage using clEnqueueWriteImage.(Note: Origin field is set to {0,0,0} and region field is set to {ImageWidth,ImageHeight,1})
7. Write the filter onto device using
8. clEnqueueWriteBuffer.
9. Create Sampler
10. Set the Global Work Size to {ImageWidth, ImageHeight}
11. Read Back the edge detected image from device to host using clEnqueueReadImage. (Note: Origin field is set to {0,0,0} and region field is set to { ImageWidth-filterWidth+1, ImageHeight-filterHeight+1,1})
12. Store the edge image and edges in separate file.

The OpenCL kernel code in Fig. 2 multiplies the given image with Laplacian of Gaussian filter to generate the convolution matrix which gives the edge-detected image.

VI. Edge Detection through Combination of OPENCL and MPI

MPI can run multiple processes simultaneously. Now those processes can be OpenCL kernel code. Thus the proposed paper achieves pixel level parallelism on multiple images in parallel. The speedup gained is N where N is the number of processes created or number of images in this case. The steps are as follows.

1) Create MPI Environment.
2) Create image number of processes.
3) Each process reads one image.
4) Each process contains OpenCL host code.
5) Each Process calls the OpenCL kernel.
6) Each Process stores the edge detected image.

The edge detected image values are also stored in excel files for further processing.

```
__kernel void
EdgeDetect(__read_only image2d_t
sourceImage,  __write_only
image2d_t outputImage, int rows,int
cols, __const float* filter,int
filterWidth,sampler_t sampler)
 {
int column=get_global_id(0);
int row=get_global_id(1);
int halfWidth=(int)(filterWidth/2);
float4 sum={0.0f,0.0f,0.0f,0.f};

int filterIdx=0;
int2 coords;
for(int i=-
halfWidth;i<=halfWidth;i++)
{
    coords.y=row+i;
    for(int j=-
halfWidth;j<=halfWidth;j++)
    {
    coords.x=column+j;
    float4 pixel;

pixel=read_imagef(sourceImage,sampl
er,coords);

sum.x+=pixel.x*filter[filterIdx++];
    }
 }
 if(row<rows && column<cols)
 {
     coords.x=column;
     coords.y=row;

write_imagef(outputImage,coords,sum
);
```

Fig. 2 OpenCL Kernel Code

VII. Results

TABLE II
Linear Relationship between Number of Processes and Number of Images with Respect to Execution Time

Number of images	Number of Processes	Time Taken
1	1	0.0001 sec
2	2	0.0001 sec
3	3	0.0001 sec
4	4	0.0001 sec
5	5	0.0001 sec
6	6	0.0001 sec
7	7	0.0001 sec

From Table II, it can be observed that a speedup of N is achieved.

TABLE III
COMPARISON OF SEQUENTIAL VERSUS PARALLEL ALGORITHM

Mode of Execution	Time Taken
Sequential	3 Sec
Parallel Using OpenCL	0.0001 Sec

From Table III, it is clear that pixel level parallelism is more efficient than its sequential counterpart. Image is taken from Sample Imagery List of GeoEye Inc. (formerly Orbital Imaging Corporation or ORBIMAGE) which is an American commercial satellite imagery company. The original image is depicted in Fig. 3. The edge detected image is depicted in Fig. 4.

Fig. 3 Original geospatial image (Image is taken from Sample Imagery List of GeoEye Inc. (formerly Orbital Imaging Corporation or ORBIMAGE) which is an American commercial satellite imagery company

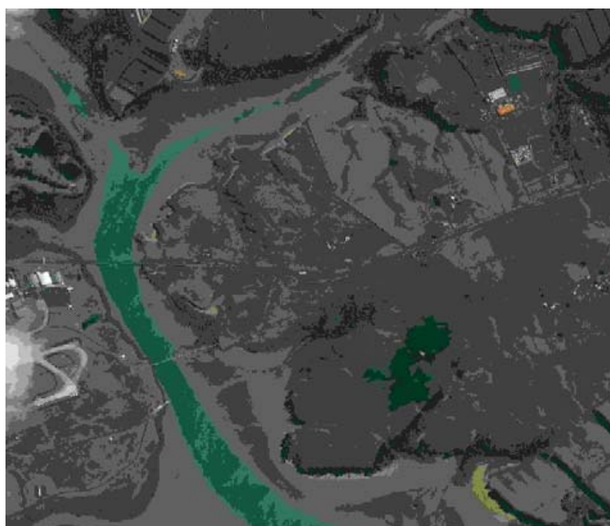

Fig. 4 Edge Detected geospatial image

VIII. CONCLUSION

The proposed work discussed the advantage of multicore and GPU implementation through MPI and OpenCL respectively. Edge detection of multiple images with pixel level parallelism is much faster than sequential counterpart. The above preprocessing step can be incorporated in parallelizing other image processing applications.

REFERENCES

[1] Benedict R. Gaster,Lee Howes, David Kaeli, Perhaad Mistry, Dana Schaa Heterogeneous Computing with OpenCLMorgan Kaufmann 2012.

[2] Raman Maini,Himanshu Aggarwal *Study and Comparison of Various Image Edge Detection Techniques.* International Journal of Image Processing (IJIP), Volume (3), Issue (1).

[3] Mohammed Baydoun, Mohamad Adnan Al-Alaoui, Rony Ferzli. Parallel Edge Detection Based on Digital Differentiator Approximation In: The 3rd International Conference on Communications and Information Technology (ICCIT-2013): Wireless Communications and Signal Processing, Beirut pp 371-375, 2013.

[4] Prakash K. Aithal, Rajesh G., Dinesh U. Acharya, MPI based edge detection of coloured image using laplacian of gaussian filter In, International Journal of Computer Application, August 2014.

[5] Varun Sanduja, Rajeev Patial Sobel Edge Detection using Parallel Architecture based on FPGA International Journal of Applied Information Systems (IJAIS) – ISSN: 2249-0868 Foundation of Computer Science FCS, New York, USA Volume 3– No.4, July 2012 – www.ijais.org.

Open Science Index, Computer and Information Engineering Vol:9, No:7, 2015 publications.waset.org/10001497.pdf