EE 451 Final Project - Spring 2021
Parallelization and Analysis of Image Processing Algorithms

Harsha Raidurgam Venkat
raidurga@usc.edu

Utkarsh Raval
uraval@usc.edu

James Meyer
jamesmey@usc.edu

Abstract—**Image processing constitutes a great share of the computational workload of current times, and with the availability of parallel processing hardware we aim to find the performance benefits of using these resources. With this project we plan to parallelize Median Filtering, Edge Detection and Image Thresholding, three very common algorithms used in image processing using OpenMP and analyze their relative performance comparing Multi-CPU and GPU implementations on AWS Cloud instances.**

1. Introduction

Image Processing, a subfield of signal processing, is a means of processing digital images to acquire, enhance, restore, compress or segment images to produce desirable output images or data from the input images, through algorithms. Images being three dimensional matrices of individual pixels and their processing algorithms being the manipulation of these matrices lend themselves to be easily parallelized. We plan to exploit this property of image processing algorithms and compare the reduced cost and increased performance of their parallel execution on Multi-CPU machine as compared to a single GPU execution.

2. Background and Motivation

With more than a billion images being captured in a day, the growing demand for reducing the execution time of image processing is ever increasing. By implementing the various parallel processing architectures learned through the coursework we plan to tackle this challenge and quantize the benefits of the parallel execution of image processing algorithms.

Our primary motivation in doing this project was the apparent advantage of parallelizing image processing algorithms, because of their simplistic matrix modification approach, which can be greatly parallelized to improve their performance. Parallel architectures can make great advantage of data parallel operations (SIMD) where the same instruction is run on multiple data, producing individual intermediate results that can then be combined to form the actual output.

Also being aware that the latest CPU architectures like the Arm-based AWS Graviton2 processors deliver up to 40% better price performance over current generation CPUs, we can push the CPU implementations to perform better or atleast in par with the GPU implementations.

3. Existing Implementation and Shortcomings

3.1. Sequential Algorithms

The traditional method to implement the sequential algorithms has been optimized to its maximum potential and would not be a feasible solution to process huge datasets as it would incur huge processing times.

3.2. OpenCV

When looking at the sequential implementation, we can notice that each pixel is computed independently. To optimize the computation, we can perform multiple pixel calculations in parallel, by exploiting the multi-core architecture of modern processors. To achieve this easily, we can use the OpenCV cv::parallel for framework. However, the user has to still explicitly parallelize the algorithms and there is no definite way defined to implement this.

3.3. CUDA

The GPU implementation could be a potential solution to optimizing these algorithms. GPUs can process images at a faster and efficient rate, however the cost factor should be considered, its implementation turns out to be more expensive as compared to cluster of CPUs. The increased overhead in data loading as well as memory management becomes a challenge when trying to optimize the CUDA implementation. The selected set of image processing algorithms do not involve complex matrix operations and hence a GPU is underutilized where these algorithms perform simple image processing operations.

4. Goal

4.1. Hypothesis

The goal is to parallelize the sequential algorithms that are most commonly used in image processing, which include

Median Filtering, Edge Detection and Image Thresholding. After coming up with the best possible approach to parallelize the algorithms we planned to compare the cost to performance ratio of the implementation by executing it on a cluster of vCPUs and executing it on a GPU. Implementing image processing algorithms on a cluster of vCPUs will allow a lower amortized cost per processed dataset compared to the algorithms on a GPU. Our point of comparison being the AWS EC2 instances that provide both cluster vCPUs as well as GPGPUs and considering the current pricings for both.

4.2. Expected Result

Our expectations predict for the cluster CPUs being the most optimal solution for low resolution ($\leq$1024px) image processing. The reason for this could be extrapolated when we try to understand why CPUs are faster than a GPU for low resolution images. CPU sets the frame up, handles all the AI/resource allocation and then passes the parameters to the GPU which then draws the frame. So the CPU reads the image/frame and sends the frame along to the GPU. The larger the frame and the more processing required the longer it takes for the GPU to finish. At low resolutions the frames are drawn much much faster therefore the CPU has to do a lot more work setting up more frames.

A fairly simple explanation is that at low resolutions, the GPU can render more frames per second. So it becomes about how quickly the CPU can send those frames to the GPU. when the CPU is the limiting factor, it becomes much easier to gauge performance.

5. Serial Algorithm

To process images within the selected category of algorithms, every single pixel generally must be transformed. With edge detection, for example, a matrix is applied to a 3 x 3 grid surrounding each pixel. The different edge detection algorithms use different matrices. For example, sobel edge detection uses xKernel = { -1, 0, 1, -2, 0, 2, -1, 0, 1 } and yKernel = { -1, -2, -1, 0, 0, 0, 1, 2, 1 }. Refer to Algorithm 1 to find the pseudocode for edge detection algorithm.

Algorithm 1 Pseudocode for Edge Detection

```
$tempImage \leftarrow image$
for $i \leftarrow 1 < image\_height - 1$ do
    for $j \leftarrow 1 < image\_width - 1$ do
        vec.clear()
        for $fx \leftarrow -1 < 1(inclusive)$ do
            for $fy \leftarrow -1 < 1(inclusive)$ do
                vec.push_back(tempImage[x + fx][y + fy])
            end for
        end for
        $gx \leftarrow inner\_product(vec, xKernel)$
        $gy \leftarrow inner\_product(vec, yKernel)$
        $image[i][j] \leftarrow abs(gx) + abs(gy)$
    end for
end for
$image \leftarrow image > threshold$
```

6. Technical Implementation

For each of the three algorithms ie... for Median Filtering, Edge Detection and Image Thresholding we are looking at has this rough structure of looping through each pixel and applying some computation to it. We see that there are a few ways of splitting up the processing work.

We are looking at two main parts of this problem to parallelize: moving images into memory and subsequently processing them. To move images into memory in parallel, we imagine a queue with producers that actually convert image files into matrices and store them in memory in the queue data structure and consumers that will pull images to be processed and saved. We may need another queue to place processed images to be saved. Benchmarking on the various potential queue styles will be done to ensure that the final one is the fastest.

For the second part, we can foresee three possible approaches to achieve parallelism in processing images by:

1) Data Parallelism
2) Task Parallelism
3) Hybrid Approach - combination of task and data parallelism

6.1. Data Parallelism

Ideally, data parallelism is a form of parallelization which relies on splitting the computation by subdividing data across multiple processors in parallel computing environments. In a multiprocessor system, data parallelism is achieved when each processor performs the same task on different pieces of distributed data.

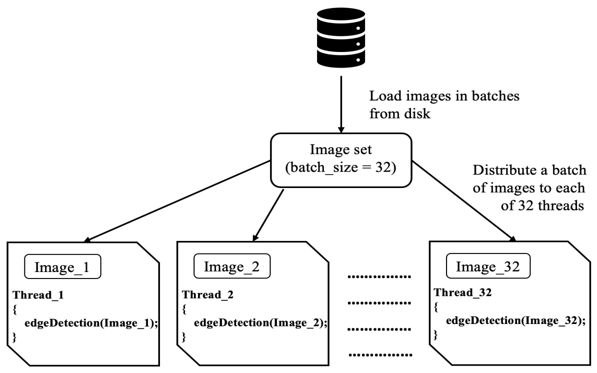

Load images in batches from disk
Image set (batch_size = 32)
Distribute a batch of images to each of 32 threads
Image_1
Thread_1 { edgeDetection(Image_1); }
Image_2
Thread_2 { edgeDetection(Image_2); }
Image_32
Thread_32 { edgeDetection(Image_32); }

Figure 1. Data Parallelism

As seen the Figure [1], we can see that instead of considering each image pixel as data, we considered each image as the data. So with a 32 vCPU machine, we can distribute 32 images simultaneously to read, process and save the processed image back to the disk. With this approach, as there is only one execution thread operating on the image completely, the speedup is more. This may make more sense because although the time it takes to process a single image may be longer, the throughput of the processing system will be higher due to decreased overhead. In order to fully utilize the CPUs, we consistently maintained 100% CPU utilization by increasing the batch_size to a higher value, so no thread is left waiting for next image. Below is the sample code snippet which distributes each image across each core.

```
int main(int argc, char **argv)
{
    int batch_size = 200;
    int num_images = 1000;
    int algo = atoi(argv[1]);
    int batches = num_images / batch_size;
    for (int i = 0; i < batches; i++)
        load_images(i * batch_size + 1,
                    (i + 1) * batch_size, algo);
}

void load_images(int start, int end, int algo)
{
    #pragma omp parallel for
    for (int i = start; i <= end; i++)
    {
      /*
        1. Load the image i from memory.
        2. Process the image based on
           algorithm id (algo) to execute.
        3. Save the processed image to memory.
      */
    }
}
```

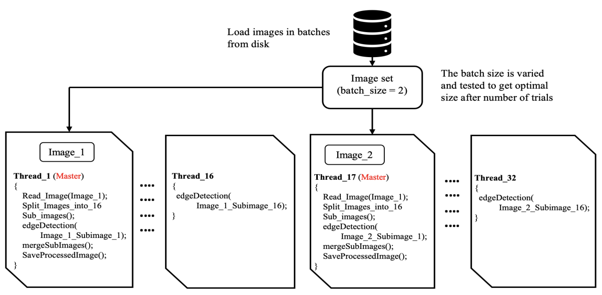
6.2. Task Parallelism

Parallel tasks do not match the execution design of the chosen algorithms since main operations such as smoothing, localization and thresholding are dependent on each other. In other words, thresholding will be applied on the results of the localization that should be smoothed previously. So, this chain of operations is required to be processed in sequence in order to provide correct output.

6.3. Hybrid Approach

On a 32 vCPU machine, two images are distributed to 16 vCPUs cores each. So, we can interpret two pairs of 16 threads executing different tasks on different data simultaneously. This can be thought as similar in lines with task parallelism.

With this approach we aim to process different parts (each block of 4*4 block) of the image individually and simultaneously. Therefore, we are required to have a data partitioning mechanism in which image is divided among the computation units. Dividing the image into suitable sized sub images will provide an independent chunk of data that can be processed in parallel. This is how data parallelism can be established in this case. Yet, one of the main challenges working on shared memory multiprocessors is the work distribution.


Load images in batches from disk
Image set (batch_size = 2)
The batch size is varied and tested to get optimal size after number of trials
Image_1
Image_2
....

```
Thread_1 (Master)
{
  Read_Image(Image_1);
  Split_Images_into_16
  Sub_images();
  edgeDetection(
      Image_1_Subimage_1);
  mergeSubImages();
  SaveProcessedImage();
}
```

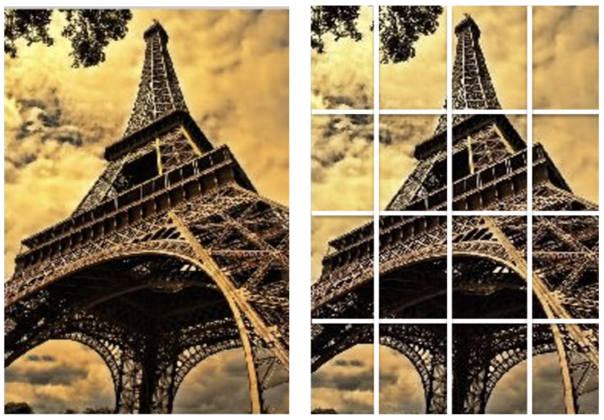
```
Thread_16
{
 edgeDetection(
      Image_1_Subimage_16);
}
```

```
Thread_17 (Master)
{
  Read_Image(Image_1);
  Split_Images_into_16
  Sub_images();
  edgeDetection(
      Image_2_Subimage_1);
  mergeSubImages();
  SaveProcessedImage();
}
```

```
Thread_32
{
 edgeDetection(
      Image_2_Subimage_16);
}
```

Figure 2. Hybrid Approach

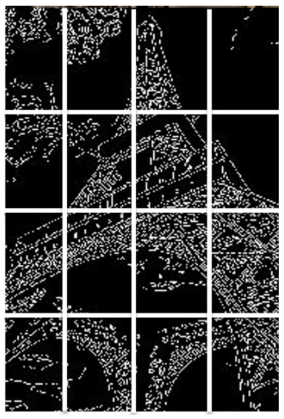
Work, for good distribution, can be divided into rows, columns or blocks as presented in Figure [2]. For instance, when we have an $M \times N$ image, the sub images are specified as follows. If the number of sub images equals x then we will have x sub images each of size $\frac{M}{X} \times N$. While partitioning into columns will lead to having x sub images each of size $M \times \frac{N}{X}$.



Partitioning the image into sub images and applying edge detection on each of the sub images: (a) Original Image (b) Divide into 4 X 4 blocks and dynamically Schedule the Sub Images among threads allocated.

Figure 3. Original and Sub-images

Partitioning into blocks results in x^2 sub images each of size $\frac{M}{X} \times \frac{N}{X}$ (as shown in Figure [3] & [4]) processing the sub images can be processed in parallel using multicore processors and then can merge the results into one final image. Using appropriate mechanisms for data partitioning not only provides an independent chunk of data that can be processed concurrently but also will add flexibility in tuning the algorithm. In particular, tuning mechanism allows the algorithm to work more efficiently on different architectural platforms which have been proven to enhance data locality which in turn reduces the number of time-consuming cache misses. Work distribution starts by allowing each of the processors to copy its assigned private data to its local memory. To gain efficiency, it is desired to decrease the number of times needed to copy data from slower shared memory to local memory at each computational unit. The algorithm


(c) Merge Sub Processed Images

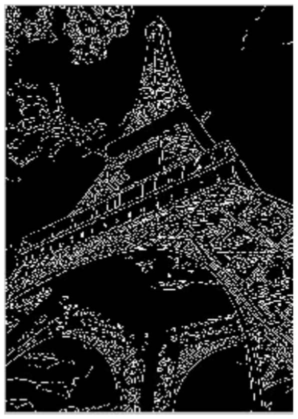

(d) Reconstruct the Processed Image.

Figure 4. Processed Sub-images and Output Image

starts dividing the image into a number of equal sized tiles (sub-images). Parallel processes will work independently on different sub-images using a self-scheduling technique for work distribution. Each processor applies smoothing, localization, and iterative thresholding before writing the processed data to its final location as shown in Figure [5].

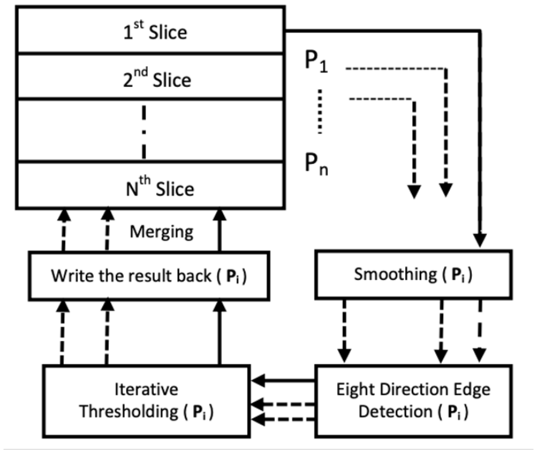

1st Slice
2nd Slice
Nth Slice
P1
Pn
Merging
Write the result back (Pi)
Smoothing (Pi)
Iterative Thresholding (Pi)
Eight Direction Edge Detection (Pi)

Figure 5. Processing life

As seen in the figure, the parallelism is applied at the highest possible level in which all work, in between the time of loading the main image up until merging sub-images, is done in parallel. The original input image is defined as an object in the code resulting in the need for some synchronization when the final result is written back to this object. This is a requirement to guarantee data integrity, therefore, each processor will write the data to its original region of interest atomically. It is best to choose self-scheduling instead of pre-scheduling for good work balance. This will prevent the problem of having some processors idle while the others have excess work due to varying amounts of work required in each region of the image.

This strategy, according to our experiments, can result in better detection where more real edges are detected since each of the sub-images can have its own optimal thresholding value. Self Scheduling will overcome the difference in time required by applying the iterative thresholding mechanism on the assigned data as the number of computations may vary depending on the data itself. The Pseudo-code of the parallel hybrid approach is shown below.

```
procedure edgeDetection()
    /* Loading the image */
    Img [imgHeight,imgWidth] := imRead[imgName];
    workLoad := imgHeight/numOfSubImage;
    shared img, destImage, numOfSubImage, levelOfNoise, numOfProc,
          toThreshold, toSmooth, workLoad, imgWidth, imgHeight;
    private i, subImg, x, y;

    /* Dynamic Scheduling of parallel processes */
    Self-Scheduled forall i := 0 until numOfSubImage

    /* partitioning data into sub-images each having the same
       imgWidth of a specific height */
    begin
        x : = 0;
        y := i * workLoad;
         /* copy specified rect to subImage */
        subImage := Rectangle( Img, Rect(x, y, imgWidth,
                                    imgHeight/numOfSubImage));
        /* De-noising   : if smoothing is enabled */
        if (toSmooth) then
            subImage := Smoothing (subImage,LevelOfNoise);
        end
        /* Localization  : Eighth direction Prewitt */
        subImage := edgeDetection (subImage);

        /* Thresholding  : if thresholding is enabled */
        if (Thresholding) then
            subImage := iterativeThresholding (subImage);
        end
        /* Merging Data  : Writing Data to its final destination*/
        critical work;
        mergeSubImage (destImage, subImage, Rect (x, y,
                     imgWidth, imgHeight /numOfSubImage ));
        end critical;
            Release(subImage);
    end
    /* Return Processed image*/
    Return (destImage);
End procedure
```

7. Execution

7.1. Dataset

We have used images from Places365 dataset. Initially, 1000 high resolution images (1024px) were downloaded. These images were resized to 512px, 256px, 128px resolution images and were used as part of this study.

7.2. Experimental Setup

Our implementations are based on C/C++ for CPU parallelization and python for testing on GPU. We employed AWS c6g.8xlarge EC2 instance to test the multi-vCPU parallelization. The platform which have been employed to test the algorithms is Arm-based AWS Graviton2 processors with 64-bit Arm Neoverse cores, 32 vCPUs, 64 GB RAM with 1 x 1900 NVMe SSD, 12Gbps network bandwidth. The best available CPU architecture and number of cores is carefully chosen in terms of cost (both time and cost per hour) to counterbalance GPU performance. For single cloud GPU testing, AWS EC2 g4dn.8xlarge is chosen. It hosts the same vCPU configuration as c6g.8xlarge. The CPUs in this intance type are 2nd Generation Intel Xeon Scalable (Cascade Lake) processors with 1 x NVIDIA T4 Tensor Core GPU, 16 GB GPU RAM.

7.3. Best Approach

The use of parallel computing lets you solve computationally and data-intensive problems using multicore processors, but, sometimes this effect on some of our control algorithm and does not give good results and this can also affect the convergence of the system due to the parallel option.

For the hybrid model, we modified the algorithms using OpenMP compiler directives for:

1) Spawning a parallel region
2) Dividing blocks of code among threads
3) Distributing loop iterations between threads
4) Serializing sections of code
5) Synchronization of work among threads

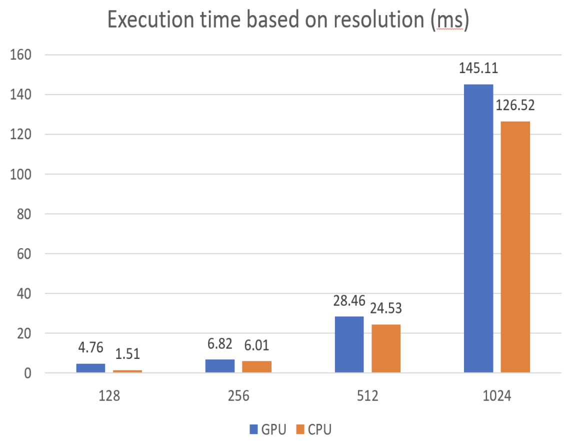
The extra cost (i.e. increased execution time) incurred are due to data transfers, synchronization, communication, thread creation/destruction, etc. These costs can sometimes be quite large, and may actually exceed the gains due to parallelization. For example, for distributing loop iterations between threads, every thread has its own execution stack that contains variables in the scope of the thread. When parallelizing code, it is very important to identify which variables are shared between the threads, and which are private. The challenges posed by the approach with the increased execution time for all the three selected algorithms was important in understanding the behavior of CPUs for this case study.

For the hybrid model, monitoring CPU utilization and performance, we observed that all the threads had more idle time and were not fully occupied to complete the tasks. This along with the above factors has contributed to higher execution times as compared with that of GPU for the hybrid model. This affected the choice of hybrid model and failed to show desirable performance that was expected. The simple data parallelism approach indeed avoids the above mentioned overheads and it was chosen to further evaluate and analyze the CPU, GPU price performance in the next section.

8. Analysis

To quantify the performance of the CPU cluster versus the GPU, we analyzed the average execution time per image and the cost per 100 GB of images fed into each algorithm for each piece of hardware. To create each graph, we ran the selected algorithm on a CPU cluster and a GPU independently. We also ran them on different image resolutions, expecting to see a trend of GPU execution times being closer to or lower than CPU execution times on higher resolution images. For each trial, we loaded and executed the algorithm on 1000 different images and averaged the results to get the time is takes per image across a large, varied dataset.

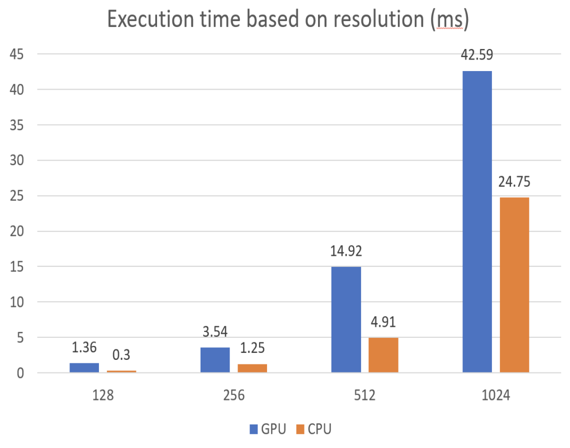
8.1. Median Filtering



Serial Execution Time: 783ms

At every image resolution, the CPU cluster has a shorter execution time than the GPU. The respective speedups at each resolution from smallest to largest is 3.15, 1.13, 1.16, and 1.15. Except for the 256 x 256 pixel (256px) resolution, the execution time ratio follows a downward trend.

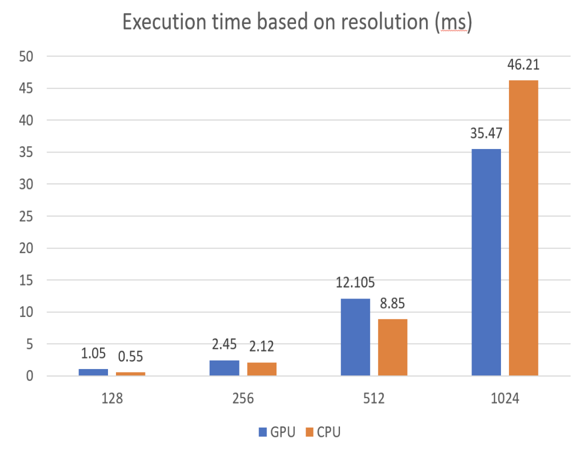
8.2. Edge Detection



Serial Execution Time: 388ms

Again, at every image resolution, the CPU cluster executes the algorithm on each image on average in less time than the GPU. However in edge detection, the speedups are more dramatic than in median filtering. At it's peak speedup, the edge detection algorithm runs 4.5 times faster on the CPU cluster than on the GPU on 128px images. The speedup again also has a downward trend as the image resolution gets higher.

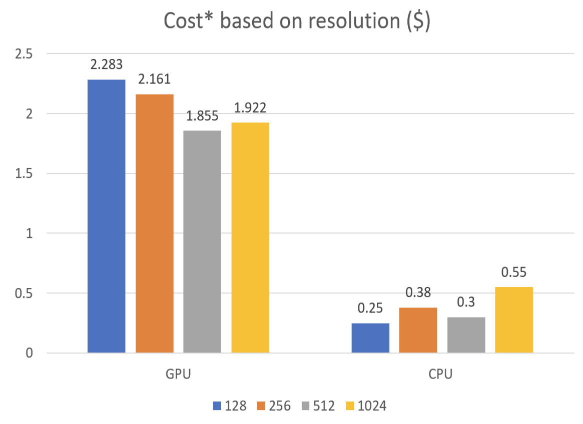
8.3. Adaptive Thresholding



Serial Execution Time: 327ms

With adaptive thresholding, the same trend of lowering speedups continues, but with images larger than 512px, the execution time for the GPU is lower than it is for the CPU cluster.

8.4. Cost of Edge Detection Comparison



*estimated cost of processing 100 GB of images at these resolutions

The cost for running the CPU cluster is $1.088 per hour, and the cost for running the GPU is $2.176 per hour. Therefore, for it to be more cost-effective to run the GPU, the GPU must run two (2) times faster than the CPU cluster on a given algorithm and dataset. Taking edge detection as an example algorithm, we can see that at all levels, the cost to run the algorithm on 100 GB of images is significantly less on the CPU cluster than it is on the GPU.

9. Insights and Conclusion

The total execution time on the CPU cluster is generally faster for all three of the algorithms. The calculated cost of execution for an image dataset of size 100 GB at a resolution of 1024px is 3.5x cheaper using the CPU cluster than using the GPU. The execution is mostly faster for images less than 1024px. Even though the CPU may take slightly more time than the GPU for greater than 1024px images, the overall cost incurred would be lower. Even in the worst case for our tested algorithms and resolutions, adaptive thresholding would cost 1.54 times less if run on the CPU cluster even though it takes 1.3 times more time than the GPU.

We can extrapolate from our data that as we increase the resolution of the images run through the algorithms, the costs for each platform would converge and the cost for the CPU cluster would eventually eclipse the cost for the GPU. While we cannot say for certain that 4096px images would run slower on the CPU cluster than on the GPU, we believe that, based on the data trends, this is likely for adaptive thresholding and plausible for the other two algorithms.

We also cannot confirm our hypothesis, but our experiments yield evidence that our hypothesis is more likely to be true. It is likely that the reason why the CPU cluster runs the algorithms faster has to do with the actual loading of the images into memory rather than the isolated execution time of the algorithms. We believe if we isolated the algorithm, thereby removing loading times, and timed it on the CPU cluster and GPU, the GPU would execute the algorithms much faster than the CPU cluster would.

Our evidence supports this hypothesis. As resolution sizes increase, the ratio of execution times between the CPU cluster and GPU gets smaller. This is what we would expect as a result of our hypothesis because as image resolutions increase, the loading times for the images make up a smaller proportion of execution time and the actual computation takes up more time. Therefore, we can extrapolate that because times are converging, it is likely due to the fact that actual computation takes much less time on the GPU than the CPU cluster.

We used OpenMP, a high-level programming model to parallelize the image processing algorithms and also explored the different approaches to achieve parallelism. In this process, we identified the challenges in adapting OpenMP for different approaches. We explored different parallelization and optimization techniques, as well as their performance impacts. The results are encouraging to indicate that the high-level, productive programming model OpenMP can be adapted to uncover the capabilities and justify CPU performance for the three image processing algorithms.

9.1. Future Work

Other DIP algorithms could be optimized to run on cloud CPU cluster for efficient and cost-effective execution for different image resolutions. Also, Open MP directives could be studied further to achieve better loop level parallelism and improve hybrid approach performance. New combinations of private and shared variables could be tried which could improve the hybrid approach performance. This would involve identification of loop parameters causing significant increase in cost and fine tuning with a alternative implementation to overcome that problem.

References

[1] Emrani, Zahra et al. “A New Parallel Approach for Accelerating the GPU-Based Execution of Edge Detection Algorithms.” Journal of Medical Signals and Sensors Vol. 7,1 (2017): 33-42.

[2] Mohammed, M.; Alaghband, G., “An Improved Parallel eight Direction Prewitt Edge Detection Algorithm,” Image Processing, Computer Vision and Pattern Recognition (IPCV’13), 2013 International Conference on , vol., no.2, 22-25 July 2013

[3] Bräunl, T., with Feyrer, S., Rapf, W., Reinhardt, M., Parallel Image Processing, Springer Verlag, Heidelberg, 2000

[4] Bräunl, T.. “Tutorial in Data Parallel Image Processing.” (2001).

[5] Palenichka, R.M. Parallel median filtering algorithms and their real-time implementation. Cybern Syst Anal 25, 694–699 (1989).

[6] Ranka, S. and S. Sahni. “Efficient serial and parallel algorithms for median filtering.” IEEE Trans. Signal Process. 39 (1991): 1462-1466.

[7] R. Mego and T. Fryza, ”Performance of parallel algorithms using OpenMP,” 2013 23rd International Conference Radioelektronika (RADIOELEKTRONIKA), 2013, pp. 236-239, doi: 10.1109/RadioElek.2013.6530923.

[8] G. Slabaugh, R. Boyes and X. Yang, ”Multicore Image Processing with OpenMP [Applications Corner],” in IEEE Signal Processing Magazine, vol. 27, no. 2, pp. 134-138, March 2010, doi: 10.1109/MSP.2009.935452.

Appendix

1. Code

Click here to view the source code on GitHub

2. Useful Links

- Digital image processing algorithms
- Edge Detection by Using Canny and Prewitt
- Open MP
- Median Filtering
- Adaptive Thresholding
- Fast and Parallel Implementation of Image Processing Algorithm on GPU using CUDA