# PARALLEL SEGMENTATION OF AN IMAGE WITH EDGE DETECTION

Katlego Penyenye

February 21, 2021

## 1 Introduction

The process of partitioning or diving a digital image into various parts or segments is known as image segmentation. Labels are assigned to all of the pixels in a digital image to segment it. These labels are assigned in such a way that pixels with the same label are comparable in some sense. A region or a segment is made up of pixels with the same label. Image segmentation is critical in computer vision because it allows computers to evaluate the content of an image and tell what is in it.Image segmentation relies heavily on edge detection, which is at the heart of many image segmentation methods, It is used to locate items inside an image and to partition an image into objects in the image and the image backdrop.Edge detection-based image segmentation has a wide range of applications in real-world challenges, for edge detection, a variety of approaches are employed [1]. The Sobel Edge Detection method will be used in this project.Applying a 3x3 mask on each pixel in the image is how the Sobel technique detects edges.This is equivalent to multiplying all the values of surrounding pixels by the mask values and adding across the mask, this process is termed convolution [2]. Convolution will be done one pixel at a time using a simple serial program. Because photos include a lot of pixels, this can take a long time. There are many parallelizations methods in this program that can be used to improve it's performance. To take use of this parallelism, we use CUDA and MPI in this project.

## 2 Methodology

There are multiple parallel solutions to most programming problems. Existing sequential algorithms may not always provide the optimal answer.The design technique we describe is designed to encourage an experimental approach to design, in which machine-independent issues like concurrency are examined early in the design process and machine-specific parts of design are left until later.Partitioning, communication, agglomeration, and mapping are the four processes in this methodology.

https://edoras.sdsu.edu/~mthomas/docs/foster/Foster_Designing_and_Building_Parallel_Programs.pdf

These four processes are described in more detail below:

1. Partitioning: Break the problem down into smaller, more precise tasks. These tasks will be carried out in parallel.
2. Communication: It is decided how much communication is required to coordinate task execution, and appropriate communication structures and algorithms are designed.
3. Agglomeration: The performance requirements and implementation costs of the task and communication structures developed in the first two stages of a design are assessed. Jobs are grouped into larger tasks if necessary to improve performance or lower development expenses.
4. Mapping: Each job is assigned to a processor in a way that tries to balance the competing goals of high processor usage and low communication costs. Load-balancing techniques can determine mapping at runtime or specify it statically.
5. The Picture below shows the methodical design:

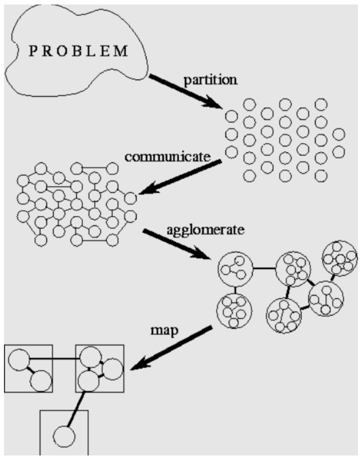

Figure 1: Methodical Design.

## 2.1 CUDA

CUDA is a parallel programming methodology that employs the graphics processing unit (GPU) to implement parallel programs. GPU contains a large number of processors, each of which may support a large number of hardware threads. The GPU is built to maximize each thread's throughput. The serial solution has the drawback of performing convolution for each pixel one at a time.Each pixel's convolution is independent of the others, allowing all pixels to be convected at the same time. Convolution for all pixels at the same time cuts down on the time it takes to segment an image using edge detection. We use CUDA programming to accomplish this, A thread can be assigned to a pixel in CUDA programming, and each thread must be assigned to all of the pixels in the image [3]. Since each pixel now has its own thread, the convolution in that thread will be handled by that thread.We implemented three CUDA programs, Global and constant, shared and constant memory and texture.

### 2.1.1 Global and Constant Memory

Variables or data that do not change are stored in constant memory. Constant memory is a read-only memory that is cheap because it is optimized for read-only use. The convolution mask will be stored in constant memory. The image will be saved in the system's global memory. The memory that can be accessed by all threads in the kernel is known as global memory.Because all threads will be accessing global memory at almost the same time, traffic will build up in global memory, slowing access.The method that is employed to resolve this conflict is also the one that causes global memory to be slow. The code snippet below uses constant and global memory to achieve image segmentation based on edge detection.

```
_global__ void __launch_bounds__(1024) convolution_global(float *image, float *output,
     size_t width){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  float val = 0;
  for (int j = 0; j < MASK_DIMENSION; ++j) {
    for (int i = 0; i < MASK_DIMENSION; ++i) {
      int d_i = i-OFFSET;
      int d_j = j-OFFSET;
      if (d_i + x >= 0 && d_i + x < width && d_j + y >= 0 && d_j + y < width) { /
        val += (float)image[y*width+x+d_j*width+d_i] * edge_kernel[j*MASK_DIMENSION+i
    ];
      }
    }
        __syncthreads();
  }
  if(val>0.4 || val<-0.4){
        output[y*width+x] = 1;
    }
    else{
        output[y*width+x] = 0;
    }
}
```

Listing 1: Global and Constant Memory

### 2.1.2 Shared and Constant Memory

As in Global and Constant Memory, we'll store the convolution mask in constant memory. Memory that can only be accessed by threads in the same block is referred to as shared memory. This means that threads from different blocks are unable to use the same shared memory. Like global memory, shared memory is a read-write memory.Instead of using global memory for threads in the same block, we can utilize shared memory. Because fewer threads are accessing shared memory, traffic is reduced. Because shared memory is a read-write memory, a system that corrects contention slows it down as well. The code snippet below uses shared and constant memory to achieve image segmentation based on edge detection.

```
_global__ void convolution_shared(float *image, float *output, size_t width){
    __shared__ float image_ds[(tile_width + MASK_DIMENSION - 1)*(tile_width +
    MASK_DIMENSION - 1)];
        int s_width = (tile_width + MASK_DIMENSION - 1);
        int ty = threadIdx.y;
        int tx = threadIdx.x;
        int block_y = blockIdx.y;
        int block_x = blockIdx.x;
        int x = block_x * tile_width + tx;
        int y = block_y * tile_width + ty;
        int h_I_t = y - MASK_OFFSET;
        int h_I_b = y + MASK_OFFSET;
        int h_I_l = x - MASK_OFFSET;
        int h_I_r = x + MASK_OFFSET;

        if (h_I_t < 0 || h_I_l < 0)
                image_ds[ty*s_width+tx] = 0;
        else
                image_ds[ty*s_width+tx] = image[y*width+x - MASK_OFFSET*width -
                MASK_OFFSET];

        if (h_I_r >= width || h_I_t < 0)
                image_ds[ty*s_width+(tx+MASK_OFFSET+MASK_OFFSET)] = 0;
        else
                image_ds[ty*s_width+(tx+MASK_OFFSET+MASK_OFFSET)] = image[y*width+x -
                MASK_OFFSET*width + MASK_OFFSET];

        if (h_I_b >= width || h_I_l < 0)
                image_ds[(ty+MASK_OFFSET+MASK_OFFSET)*s_width+tx] = 0;
        else
                image_ds[(ty+MASK_OFFSET+MASK_OFFSET)*s_width+tx] = image[y*width+x +
                MASK_OFFSET*width - MASK_OFFSET];

        if (h_I_r >= width || h_I_b >= width)
                image_ds[(ty+MASK_OFFSET+MASK_OFFSET)*s_width+(tx+MASK_OFFSET+
                MASK_OFFSET)] = 0;
        else
                image_ds[(ty+MASK_OFFSET+MASK_OFFSET)*s_width+(tx+MASK_OFFSET+
                MASK_OFFSET)] = image[y*width+x + MASK_OFFSET*width +
                MASK_OFFSET];

        __syncthreads();

        float out = 0;
        for (int j = 0; j < MASK_DIMENSION; ++j) {
                for (int i = 0; i < MASK_DIMENSION; ++i) {
                        out += (float)image_ds[(j + ty)*s_width+(i + tx)] *
                        const_edge[j*MASK_DIMENSION+i];
                }

        }
    if(out>0.4 || out<-0.4){
        output[y*width+x] = 1;
    }
    else{
        output[y*width+x] = 0;
         }

}
```

Listing 2: Shared and Constant Memory

### 2.1.3 Texture Memory

Another type of read-only memory is texture memory. We can use texture memory to store the image because it is significantly larger than constant memory. Texture memory is a read-only memory, it is optimized for read-only operations because no contention may occur.For the 2D special locality, the texture cache has also been optimized. Due to the fact that our image is in the form of a 2D matrix, it's a significant benefit for us. When readings have certain access patterns, texture memory can minimize memory traffic.The code snippet below uses Texture memory to achieve image segmentation based on edge detection.

```
__global__ void conv_tex(float *out,int width,int height,int m_width,int m_height){
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    int offset = m_width/2;
    float value = 0;
    for (int j = 0; j < m_height; ++j) {
            for (int i = 0; i < m_width; ++i) {
                    value += tex2D(tex,y-(int)(m_width/2)+i ,
                    x-(int)(m_height/2)+j)*tex2D(tex_edge,i,j);
            }
        }
    if(value>0.4 || value<-0.4){
        out[x*width+y] = 1;
    }
    else{
        out[x*width+y] = 0;
    }

}
```

Listing 3: Texture Memory

## 2.2 MPI

MPI is a message-passing-based parallel programming model. MPI, like CUDA, parallelizes a program by having multiple processors or processing components working on it.In contrast to CUDA, where all processing elements share memory, MPI gives each processing element or processor its own memory. This means that there are no memory traffic or contention issues with MPI.As a result, we can eliminate the overheads of memory traffic and conflict correction mechanisms by using MPI. Because each processor has its own memory, the processing parts must convey data to one another in order to communicate. This has an overhead that was not present in CUDA. MPI provides functions for data communication to reduce the overhead of data communication.The features have been designed with data communication in mind. The convolution mask will be used by all processors because it is the same and does not change. The image will only be loaded by one processor. MPI Scatter is then used to send distinct rows to each processor. On its sub-image, the CPU will then conduct a convolution.When all of the processors are finished, we use MPI Gather to collect all of the separate results into one processor.The code snippet below uses MPI to achieve image segmentation based on edge detection.

```
MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&p_s_id);
    MPI_Comm_size(MPI_COMM_WORLD,&p_s);

    runtime = -MPI_Wtime();

    if(p_s_id==0){
        data = stbi_load("data/lena_bw.pgm", &width, &height, &channel,0);
        if(data == NULL){
            printf("unable to load data\n");
            exit(0);
        }
    }

    MPI_Bcast(&width, 1, MPI_INT, 0,MPI_COMM_WORLD);
    MPI_Bcast(&height, 1, MPI_INT, 0,MPI_COMM_WORLD);

    l_w = width/p_s;
    l_h = height/p_s;

    unsigned char *image = (unsigned char*)malloc(sizeof(unsigned char)*l_h*width);
    unsigned char *output = (unsigned char*)malloc(sizeof(unsigned char)*l_h*width);

    MPI_Scatter(data,l_h*width,MPI_UNSIGNED_CHAR,image,l_h*width,MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
    ///Image segmentation part
    for (int y = 0; y < l_h; ++y) {
                for (int x = 0; x < width; ++x) {
                        int offset = MASK_DIM/2; //kernel offset with integer division
                        float value = 0;
                        for (int j = 0; j < MASK_DIM; ++j) { //perform convolution on a pixel
                                for (int i = 0; i < MASK_DIM; ++i) {
                                        int mapi = i-offset;
                                        int mapj = j-offset;
                                        if (mapi + x >= 0 && mapi + x < width && mapj + y >= 0 && mapj + y < width) { //check if in bounds of image
                                                value += ((float)image[y*width+x+mapj*width+mapi]/255) * kernel[j*MASK_DIM+i];
                                        }
                                }
                        }if(value>0.4 || value<-0.4){
                output[y*width+x] = 255;
            }else{
                output[y*width+x] = 0;
            }
        }
    }
    MPI_Gather(output, l_h*width,MPI_UNSIGNED_CHAR,data, l_h*width,MPI_UNSIGNED_CHAR, 0,MPI_COMM_WORLD);
    runtime += MPI_Wtime();
    if(p_s_id==0){
        stbi_write_bmp("data/lena_bw_mpi.bmp",width,height,channel,data);
        printf("wrote to data/lena_bw_mpi.bmp\n");
        printf("Total elapsed time: %10.6f\n", runtime);
        stbi_image_free(data);
    }
    MPI_Finalize();
```

Listing 4: MPI

## 3 Experimental setup.

We will test each performance of all the programs we implement in the Mathematical science lab cluster.All implementations will utilize the same image, and we will only use one image. All of the programs will be timed, and their results will be determined by how long it took them to finish the task.The SDK timer will be used to track the duration of all CUDA implementations. MPI timer functions will be used to calculate the duration of the MPI implementation. All implementations must outperform the serial implementation.The serial version will be implemented, timed, and used as a benchmark for our parallel programs. We use the Sobel Edge Detection method for edge detection. Below is the image used for all implementations and the convolution mask for the Sobel Edge Detection method.

We've also included the serial implementation, which will serve as the project's benchmark.Below is the serial code snippet.

|  |  |  |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

Table 1: Sobel Edge Mask.

Figure 2: Sequential Image.

```
float* serial_convl(float* image,float *kernel, size_t width,int height) {
  float* output = (float*)malloc(sizeof(float)*width*height);
  for (int y = 0; y < width; ++y) {
    for (int x = 0; x < width; ++x) {
      float val = 0;
      for (int j = 0; j < MASK_DIMENSION; ++j) {
        for (int i = 0; i < MASK_DIMENSION; ++i) {
          int d_i = i-OFFSET;
          int d_j = j-OFFSET;
          if (d_i + x >= 0 && d_i + x < width && d_j + y >= 0 && d_j + y < width) {
           val += (float)image[y*width+x+d_j*width+d_i] * kernel[j*MASK_DIMENSION+i];
          }
        }
      }
            if val>0.4 || val<-0.4){
                output[y*width+x] = 1;
            }
            else{
                output[y*width+x] = 0;
            }
        }
    }
    return output;
 }
```

Listing 5: Serial Implementation

## 3.1 Results

As shown in the table below, all of the parallel applications performed better than the serial implementation, this suggests that we were able to take advantage of the parallelism. The MPI implementation performed better than all the implementations.

### 3.1.1 The Image result by all Implementations

Please refer to the last folder (6) i.e Results, inside the source code folder.

| Implementation | Performance in milli sec(ms) |
|---|---|
| Serial | 14.463000 |
| Global and Constant Memory | 0.147000 |
| Shared and Constant Memory | 0.119000 |
| Texture Memory | 0.076000 |
| MPI | 0.016164 |

Table 2: Results.

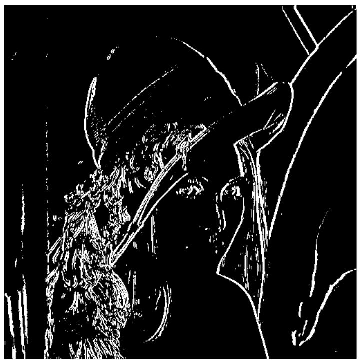

Figure 3: Output by all implementations

## 4 Conclusion

We used CUDA and MPI as parallel programming models to create parallel algorithms for picture segmentation based on edge detection in this research. Our parallel programs were created using Foster's Design Methodology. We used CUDA programming to create three parallel programs.These three applications used CUDA memory hierarchy that included global, shared, constant and texture memory. All of the CUDA implementations failed to match texture memory's performance. The MPI technique performed better than all CUDA implementations.We saw that image segmentation based on edge detection benefits from parallel programming, and the MPI technique provides the best performance than any other implementation in this paper.

## References

[1] Hou, Zujun J., and Guo-Wei Wei. "A new approach to edge detection." Pattern Recognition 35.7 (2002): 1559-1570.

[2] Podlozhnyuk, Victor. "Image convolution with CUDA." NVIDIA Corporation white paper, June 2097.3 (2007).

[3] Cook, Shane. CUDA programming: a developer's guide to parallel computing with GPUs. Newnes, 2012.