ForensicVision.docx

by Aryan Singh 220905492

Submission date: 05-Apr-2025 11:16AM (UTC+0800)

Submission ID: 2635691302

File name: ForensicVision.docx (1.58M)

Word count: 1867

Character count: 9527

ForensicVision: GPU-Accelerated Image Enhancement

Problem Statement

Crime investigations often rely on images captured from CCTV footage, crime scenes, or fingerprint scans. However, these images are frequently degraded by noise, low resolution, motion blur, or poor lighting, making it difficult to extract crucial details. Traditional image processing techniques on CPUs can be slow and inefficient for large datasets.

Introduction

ForensicVision is a CUDA-based image processing application designed to enhance degraded images using GPU acceleration. The system implements several image enhancement techniques, including rotation, Gaussian blur, edge detection, median filtering, and morphological operations. By leveraging parallel computing power, ForensicVision significantly reduces processing time compared to conventional CPU-based approaches.

Features

1. Image Rotation

- Rotates images by a user-defined angle.
- Uses trigonometric transformations for accurate pixel repositioning.

2. Gaussian Blur

- Smooths images to reduce noise.
- Uses a Gaussian kernel for convolution.

3. Edge Detection (Sobel Operator)

- Detects edges using horizontal and vertical gradient calculations.
- Utilizes Sobel kernels to highlight image contours.

4. Median Filter (Noise Reduction)

- Removes salt-and-pepper noise.
- Uses a sliding window to replace each pixel with the median of its neighbors.

5. Morphological Dilation

- Expands bright regions in binary or grayscale images.
- Enhances object visibility and fills small gaps.

6. Morphological Erosion

- Shrinks bright regions.
- Removes small noise particles and refines edges.

//implementation

1- Image rotation

```
//cuda kernel for image rotation
__global__ void img_rotate(DATATYPE *dest_data , DATATYPE *src_data , int W , int H , float sinTheta , float
cosTheta)
{
  //finding the row and column index
  int ix = threadIdx.x; //column index
  int iy = threadIdx.y; //row index
  //computing the center of the image
  float x0 = W/2.0f;
  float y0 = H/2.0f;
  //shifting the origin to image center
  float xOff = ix - x0;
  float yOff = iy - y0;
  //applying rotation transformation
  int xpos = (int)(xOff * cosTheta + yOff * sinTheta + x0);
  int ypos = (int)(yOff * cosTheta - xOff * sinTheta + y0);
  //checking for the validation and if they are valid we copy the pixel
  if(xpos>=0 && xpos<W && ypos>=0 && ypos<H)
  {
    dest_data[iy*W+ix] = src_data[ypos*W+xpos];
  }
  else
  {
    dest_data[iy*W+ix] = 0;//the out of bond pixels are set to black
  }
}
```

2- Image convulation

```
3-  //cuda kernel for convulation
4-  //performing image convulation using a kernel which act as filter
```

```
__global__ void convulation(DATATYPE *dest_data , DATATYPE *src_data , float *kernel , int W , int H , int
kernelSize)
{
    //locating each pixel with the help of grid and block indexing
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iy = blockIdx.y*blockDim.y+threadIdx.y;
    if (ix < W && iy < H) {//processing of the image within the bounds
      float sum = 0.0f;
      int halfKernel = kernelSize / 2;//helps in centering of  the kernel over the pixel
     //iteration over the kernel window
      for (int ky = -halfKernel; ky <= halfKernel; ky++) {
          for (int kx = -halfKernel; kx <= halfKernel; kx++) {
              int x = ix + kx;
              int y = iy + ky;
              //performing the weighted sum operation
              if (x >= 0 && x < W && y >= 0 && y < H) {
                  sum += src_data[y * W + x] * kernel[(ky + halfKernel) * kernelSize + (kx + halfKernel)];
              }
          }
      }

      //modifying the values so that ir stays in the range
      if (sum < 0) sum = 0;
      if (sum > 65535) sum = 65535;

      dest_data[iy * W + ix] = (DATATYPE)sum;//storing the value in the output image
  }
}
```

3- cuda kernel for median filtering

```
//cuda kernel for median filtering(removes the noice by replacing each filter with the median of its neighborhood )
__global__ void medianFilter(DATATYPE *dest_data, DATATYPE *src_data, int W, int H, int windowSize) {
    //computing the global thread index
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;

    if (ix < W && iy < H) { //checking if the the pixel is within image bound
        //finding out the half width of the window as the median operates on a square window
        int halfWindow = windowSize / 2;
        DATATYPE values[MAX_KERNEL_SIZE * MAX_KERNEL_SIZE]; //storing the pixel values from the
neighbourhood
        int count = 0;

        //collecting pixels from the neighbourhood
        for (int ky = -halfWindow; ky <= halfWindow; ky++) {
            for (int kx = -halfWindow; kx <= halfWindow; kx++) {
```

```
                //computing the actual coordinates of the neighbourhood
                int x = ix + kx;
                int y = iy + ky;
                //storing the valid pixels values
                if (x >= 0 && x < W && y >= 0 && y < H) {
                    values[count++] = src_data[y * W + x];
                }
            }
        }

        //sorting using bubble sort(suitabe for small window size due to O(n2) complexity)
        for (int i = 0; i < count-1; i++) {
            for (int j = 0; j < count-i-1; j++) {
                if (values[j] > values[j+1]) {
                    DATATYPE temp = values[j];
                    values[j] = values[j+1];
                    values[j+1] = temp;
                }
            }
        }

        //setting the median values for the output image
        dest_data[iy * W + ix] = values[count / 2];
    }
}
```

4- cuda kernel for morphological dilation

```
//cuda kernel for morphological dilation
//it performs dilation (shrinking bright regions)
__global__ void dilate(DATATYPE *dest_data, DATATYPE *src_data, int W, int H, int kernelSize) {//here the kernel
size refers to square window for dilation
    //computing global index
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;

    if (ix < W && iy < H) {//ensuring the valid image boundaries
        DATATYPE maxVal = 0;
        int halfKernel = kernelSize / 2;//computing the half size of the kernel window which helps in centering around the
current pixel
        //iterating over the kernel window
        for (int ky = -halfKernel; ky <= halfKernel; ky++) {
            for (int kx = -halfKernel; kx <= halfKernel; kx++) {
                int x = ix + kx;
```

```
            int y = iy + ky;
            //ensuring whether the range
            if (x >= 0 && x < W && y >= 0 && y < H) {
                //if the neighbour is brighter update maxVal(keep tracking of the maximum value in the neighbourhood)
                if (src_data[y * W + x] > maxVal) {
                    maxVal = src_data[y * W + x];
                }
            }
        }
    }
    //storing hte maximum pixel value in the corresponding position in des_data
    dest_data[iy * W + ix] = maxVal;
  }
}
```

5- morphological erosion

```
//cuda kernel for morphological erosion
//it shrinks the brighten regions
__global__ void erode(DATATYPE *dest_data, DATATYPE *src_data, int W, int H, int kernelSize) {
  //calculating absolute pixel position
  int ix = blockIdx.x * blockDim.x + threadIdx.x;
  int iy = blockIdx.y * blockDim.y + threadIdx.y;

  if (ix < W && iy < H) {
    DATATYPE minVal = 65535; // initialising the minimum value
    int halfKernel = kernelSize / 2;//computing the half size of the kernel window which helps in centering around the
current pixel

    for (int ky = -halfKernel; ky <= halfKernel; ky++) {
        for (int kx = -halfKernel; kx <= halfKernel; kx++) {
            //calculation of the neighbour coordinates
            int x = ix + kx;
            int y = iy + ky;
            //updating the minVal
            if (x >= 0 && x < W && y >= 0 && y < H) {
                if (src_data[y * W + x] < minVal) {
                    minVal = src_data[y * W + x];
                }
            }
        }
    }
    //storing the minVal in the corresponding dest_data , completing the erosion process
    dest_data[iy * W + ix] = minVal;
  }
```

```
}
```

6- Creating a gaussian kernel

```
//funciton to create Gaussian Kernel can be used for blurring and edge detection
void createGaussianKernel(float *kernel, int kernelSize, float sigma) {
    int halfKernel = kernelSize / 2;
    float sum = 0.0f;
    //computing the gaussian values
    //shifting of the x and y so that kernel index is positive
    for (int y = -halfKernel; y <= halfKernel; y++) {
        for (int x = -halfKernel; x <= halfKernel; x++) {
            float value = expf(-(x*x + y*y) / (2.0f * sigma * sigma));
            kernel[(y + halfKernel) * kernelSize + (x + halfKernel)] = value;
            sum += value;
        }
    }

    // Normalize the kernel in order to prevent brightness changes when applied to kernel
    for (int i = 0; i < kernelSize * kernelSize; i++) {
        kernel[i] /= sum;
    }
}
```

7- Creating a sobel kernel(edge detection)

```
// Function to create edge detection (Sobel) kernel
void createSobelKernel(float *kernelX, float *kernelY, int kernelSize) {
    //kernelX and kernelY to store horizontal and vertical Sobel respectively
    // Horizontal Sobel kernel - penalizing the pixels in the left , highlighting the piexels in the right  and keeping the
center unchanged , emphasizing horizontal changes in the intensity from dark to light
    kernelX[0] = -1.0f; kernelX[1] = 0.0f; kernelX[2] = 1.0f;
    kernelX[3] = -2.0f; kernelX[4] = 0.0f; kernelX[5] = 2.0f;
    kernelX[6] = -1.0f; kernelX[7] = 0.0f; kernelX[8] = 1.0f;

    // Vertical Sobel kernel - penalizes the pixes above and hight lighting the pixels below keeping the center
unchanged emphasizing on vertical changes of the intensity
    kernelY[0] = -1.0f; kernelY[1] = -2.0f; kernelY[2] = -1.0f;
    kernelY[3] = 0.0f;  kernelY[4] = 0.0f;  kernelY[5] = 0.0f;
    kernelY[6] = 1.0f;  kernelY[7] = 2.0f;  kernelY[8] = 1.0f;
}
```

6- function to read the pgm file

```
// Function to read PGM file and stores its information in the dynamically allocated structure
struct PGMstructure* readPGM(const char* filename) {
    FILE *imagein; //pointer to store the reference to the opened PGM file
    int row, col;
    unsigned int ch_int; //temporary variable to store the pixel values

    struct PGMstructure *imginfo = (struct PGMstructure *)malloc(sizeof(struct PGMstructure));//dynamically allocated memory to store PGM image details
    //opening the fle
    imagein = fopen(filename, "r");
    if (imagein == NULL) {
        printf("Error opening file %s\n", filename);
        free(imginfo);
        return NULL;
    }

    char magicNumber[3];
    fscanf(imagein, "%s", magicNumber);//reading the PGM header
    fscanf(imagein, "%d %d", &imginfo->width, &imginfo->height);//reading the width and height
    fscanf(imagein, "%d", &imginfo->maxVal);//reading the maximum grayscale value from the file
    //allocating memory for image data
    imginfo->data = (DATATYPE *)malloc(imginfo->width * imginfo->height * sizeof(DATATYPE));
    //reading the pixel data
    for (row = 0; row < imginfo->height; row++) {
        for (col = 0; col < imginfo->width; col++) {
            fscanf(imagein, "%u", &ch_int);
            imginfo->data[row * imginfo->width + col] = ch_int;
        }
    }
    fclose(imagein);

    return imginfo;//returning the image structure
}
```

7- function to write to pgm file

```
//function to write to pgm file
void writePGM(const char* filename, struct PGMstructure* img, DATATYPE* data) {
    FILE *imageout;//pointer to the output file
    int row, col;
    //opening the output file
    imageout = fopen(filename, "w");
    if (imageout == NULL) {
```

```
        printf("Error opening output file %s\n", filename);
        return;
    }
    //write the pgm header ,row , column and maximum greyscale value
    fprintf(imageout, "P2\n%d %d\n%d\n", img->width, img->height, img->maxVal);
    //writing the pixel data
    for (row = 0; row < img->height; row++) {
        for (col = 0; col < img->width; col++) {
            fprintf(imageout, "%d ", data[row * img->width + col]);
        }
        fprintf(imageout, "\n");
    }

    fclose(imageout);
}
```

Memory Management

- **Host-to-Device Transfer:** Copies image data to GPU memory before processing.
- **Device-to-Host Transfer:** Retrieves processed image data from GPU memory.
- **Dynamic Memory Allocation:** Allocates memory for input images, output images, and filter kernels.

Input and Output

- **Input:**
  - PGM (Portable Gray Map) images.
  - User-defined parameters (e.g., rotation angle, kernel size).
- **Output:**
  - Enhanced images saved in PGM format.

//Sample input output

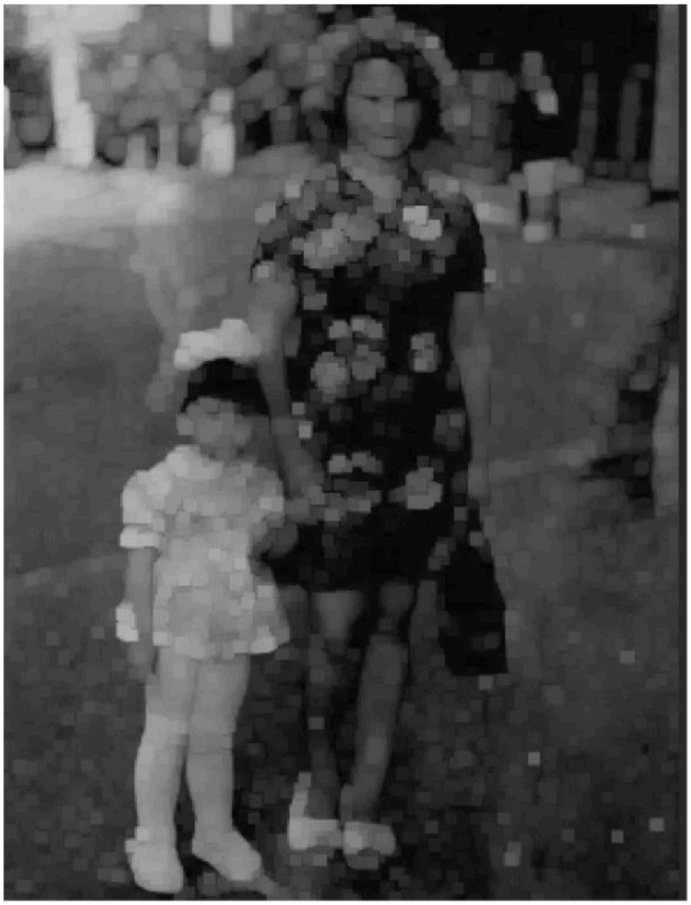
1- Input

```
Enter PGM file path: balloons_noisy.ascii.pgm

ForensicVision: GPU-Accelerated Image Enhancement
1. Image Rotation
2. Gaussian Blur
3. Edge Detection (Sobel)
4. Median Filter (Noise Reduction)
5. Morphological Dilation
6. Morphological Erosion
Choose operation (1-6): 4
Enter window size (odd number, e.g., 3, 5): 7
Enter output file path: outputnew.pgm
Processing complete. Output saved to outputnew.pgm
```

//output

2-input



```
ForensicVision: GPU-Accelerated Image Enhancement
1. Image Rotation
2. Gaussian Blur
3. Edge Detection (Sobel)
4. Median Filter (Noise Reduction)
5. Morphological Dilation
6. Morphological Erosion
Choose operation (1-6): 6
Enter structuring element size (odd number, e.g., 3, 5): 5
Enter output file path: pleasework.pgm
```

//output

ForensicVision.docx

ORIGINALITY REPORT

10%	10%	8%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	gist.github.com Internet Source	4%
2	forums.developer.nvidia.com Internet Source	3%
3	file.allitebooks.com Internet Source	3%

Exclude quotes	Off	Exclude matches	< 3%
Exclude bibliography	On		