Checking Multi-GPU Parallelism for 3D tomographic re-construction of neutron projection images of plant root samples in soil

Jenia Afrin Jeba

Student ID: 11259150, NSID: jaj212

1. Introduction

For the implementation project of this course I decided to work on an existing 3D image reconstruction application "3D tomographic re-construction of a series of neutron projection images of plant root samples in soil" which was developed by the researchers from image lab of our university [1]. Three-dimensional (3D) reconstruction of neutron tomographic projection images are an important tool for research on animal and plant tissues. Neutron scattering images contain impulsive noise caused by background cosmic gamma radiation that can significantly affect the reconstruction quality. Common de-noising methods are computationally efficient but may also blur edges in the signal reducing the quality of reconstruction and may require careful parameter selection. Moreover, prior to reconstruction for rotation axis misalignment during data acquisition and suppress statistical noise due to variations in the neutron source must be corrected [1]. Currently many of these steps require manual intervention and parameter selection to maximize reconstruction quality.

This whole process is divided into multiple stages, each of them involves multi-core CPU processing which performs in an acceptably good speed on reconstructing an individual image, but question of performance arises when there are a lot of images to be processed as a whole. A significant amount of time consumption can be saved if some of the stages can be executed in multi-core GPUs facilitating core level parallelism [4].

So, I in this implementation project I want to explore GPU-level parallelism for this image reconstruction process by implementing CUDA version of the existing application with the main motivation to bring the total processing time of running the application down to a reasonable amount. My primary goal is to provide a comparative analysis of the time consumption of the same application into multi-core CPUs vs. multi-core GPUs.

The organization of this report is as follows: a brief definition of the problem statement is given in Section 2; Section 3 describes the dataset and a brief explanation of the flow of the image processing; then an in-depth overview about the implementation of code is given in Section 4; after that, a description about experimental setup, evaluation and results is given in Section 5; lastly, Section 6 concludes this report.

2. Problem definition

My main concern is to reduce the time consumption taken by the first two stages of this image processing application namely – spike noise removal and bright field correction, with the help of multi-core GPU programming in PyCUDA. Since the application is already built in Python programming language thus, I decided to implement the application in PyCUDA (Python libraries of GPU programming language CUDA) [2][3]. A programming library called openCV was also used which is considered as one of the well-defined libraries in the field of computer vision. It offers C++, C, Python and Java interfaces and supports almost all the operating system platforms [7][9]. So, for the python interface, the computation is done via a python wrapper which wraps the original library written in C/C++. OpenCV also support GPU programming. Its GPU module is written using NVIDIA CUDA runtime API that offers even more computational power [10][13].

3. Dataset Acquisition and Image Processing Pipeline

3.1 Dataset Acquisition

Plant roots were imaged intact using a thermal neutron beam produced by the N5 Triple Axis Spectrometer at the Canadian Neutron Beam Centre, which operates a

series of neutron beamlines at the Canadian Nuclear Laboratories in Chalk River, Ontario. Plants were raised in a variety of soil media in commercial aluminum beverage containers. The beam size at the sample position on N5 in imaging mode allows the entire root system to be imaged in a single exposure for juvenile plants. Initial images were acquired using aluminum containers and sand as the dominant soil medium, since both of these materials are essentially transparent to thermal neutrons [1].

Plant containers were approximately centered in the neutron beam using crossed linear motion stages (one vertical and one horizontal) with motion step size resolution of about a micron and then were rotated through at least 180 degrees, with a minimum of one pair of exposures that were exactly 180 degrees apart. Most commonly, exposures were taken every degree or every two degrees through this range. Neutrons were detected using an Andor Ikon L 936 camera.

3.1. Image Processing Pipeline

Figure 1 shows the steps needed to produce a 3D volume from the set of projection images. There is a series of preprocessing steps, followed by noise reduction, and finally the 3D reconstruction [1].

The pre-processing begins with the removal of bright noise spikes caused by background cosmic radiation (gamma) sources, which are external to the experimental environment, in both the sample images and the bright field (BF) image. Then sample images are normalized using the BF image to correct for variations in total neutron flux across the detector. The sample images are then transformed to horizontally center the axis of rotation and ensure that it is vertical.
In the statistical noise reduction step the images are processed with a total variation minimization algorithm to reduce the overall noise level. This enhances the clarity and quality of the images and preserves strong edges. Finally, publicly available software is used to perform forward and back-projection to reconstruct a 3D volume for the sample.
I worked on parallelizing the first two steps of this pipeline namely removal of noise spikes and bright field correction in multi-core GPUs.

3.1.1 Spike Noise Removal

Spike noise pixels, also known as gamma spots, result from background cosmic radiation interacting with the neutron detector. Noisy pixels typically have much higher brightness than surrounding pixels which has deleterious effects on 3D reconstruction and further image analysis. The very bright nature of the spikes, comprised of compact groups of pixels, makes them difficult to remove with discontinuity preserving smoothing filters (e.g. total variation minimization, non-local means filter, block matching 3D filter) because they are interpreted as important strong edges and retained. Hindasyah et al. developed a method combining mean harmonic filtering with adaptive threshold median filtering to remove the spikes [1].

3.1.2 Neutron Bright Field Correction

Plant samples are placed between the neutron source and the CCD area detector. The neutron generator is approximately a point neutron source and the neutron beams are emitted divergently. Thus, the detector units closer to the center of the CCD area detector will receive more neutrons than those which are near the edges of the detector. This is evident from the bright field image in Figure 2(a). To normalize for this effect, the dataset provides 10 BF images with 60 second exposure which are averaged to obtain a single BF image estimate. The BF image pixel values can be interpreted as the maximum possible observable neutron count for each detector unit during the exposure time. This simultaneously accounts for detector efficiency and neutron flux inhomogeneity. The BF images contain spike noise just as the sample images do. The spike noise removal process for BF images is the same as for the sample images in each pipeline. The purpose of performing BF correction is to remove the effect of uneven neutron exposure across the CCD detectors.
Letting I(x, y) be the original image, and BF(x, y) the bright field image, the BF-normalized image N(x, y) is defined as: $N(x, y) = \min(I(x, y)/BF(x, y), 1)$
The minimum of the ratio and 1 is required since the BF image is an average and detector units may detect slightly more than the estimated maximum in the BF image resulting in ratios slightly more than 1.
Figure 2 shows an example BF correction. Figure 2(a) is a BF image after processing to remove spike noise.

Figure 2(b) is a sample image after spike noise removal, and Figure 2(c) is the result of BF correction of Figure 2(b).

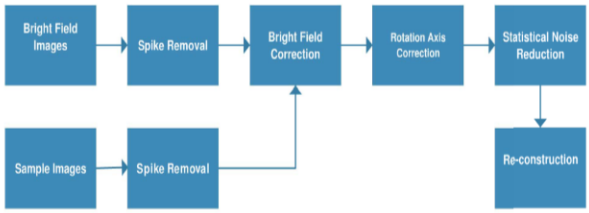


Figure 1. Pipeline of automatic noise reduction process on neutron plant root tomographic images

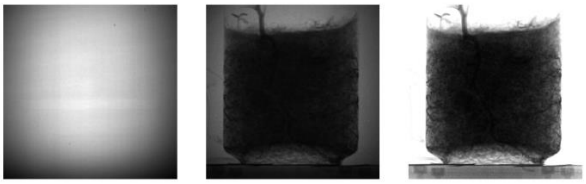


Fig 2: (a) is a sample BF image (spike noise removed) with contrast enhanced for better visualization. (b) is a sample image (spike noise removed). (c) is the BF corrected image

4. Implementation of Code

There are readily available and standardized Python libraries, such as PyCUDA and Scikit-CUDA, which make GPU programming more readily accessible to aspiring GPU programmers.

4.1 Code implementation

After learning about how the median filter works on spike noise removal, I found these python libraries useful: numpy - used as an efficient multi-dimensional container of generic data, can define arbitrary data-types, can declare powerful N-dimensional array object etc. [16]; from skimage.restoration module imported inpaint where inpainting is the process of reconstructing lost or deteriorated parts of images and videos [17]; scikit-image – it is an image processing Python package that works with numpy arrays [19].

The most important thing for removing spike noise is the use of a median filter which is a non-linear digital filtering technique that it preserves edges while removing noise [18]. The median filter is supposed to function in this way: placed over an image, run through the whole image pixel by pixel and replacing each pixel's value with the median of the neighboring pixels. First, all the pixel values from the window is sorted into numerical order for calculating the median, after that the pixel being considered is replaced with the median pixel value [18].

The first step in the image application that has been considered here, was apply filter to smooth the image out using a 2D filter after converting the image into float32 type and then compute the percentile [7], `np.zeros` has been used to create a blank 1D array to be initialized being filled with zeros one for holding the minimum value and another for holding the maximum value from the value_buffer to be used later; after that `cuda.memcpy_dtoh` function has been used to transfer data between the host (CPU) and device (GPU) [4][6].

```
def percentile(image_buffer, q):

    find_value(_image_max, image_buffer)
    max = np.zeros(1, np.float32)
    cuda.memcpy_dtoh(max, _value_buffer)

    find_value(_image_min, image_buffer)
    min = np.zeros(1, np.float32)
    cuda.memcpy_dtoh(min, _value_buffer)


    less = np.zeros(1, np.int32)
    value = (min + max) / 2

    while True:
        cuda.memcpy_htod(_value_buffer, np.int32(0))
        _percentile(image_buffer, value,
_value_buffer, block = (_work_group_size, 1, 1),
grid = (_grid_size, 1))
        cuda.memcpy_dtoh(less, _value_buffer)
        diff = q / 100 - less / _image_size


        if abs(diff) < 0.000001:
            return value


        if diff > 0:
            min = value
            value = (value + max) / 2
        else:
            max = value
            value = (min + value) / 2


def do(img_list, min, max, avg_window_size,
select_window_size, bright_average, ipmask, out_bf,
fname, fmt_out):
    if not(len(img_list)):
        return
```

After that image ratio between original and filtered image has been calculated and following conditions has

been applied to set the min / max values and collect the value (as directed by the original python code):

```
def find_value(function, image_buffer):
    gs = _image_size
    i = 0

    while True:
        if i == 0:
            inb = image_buffer
            outb = _tmp0_buffer
        elif i % 2:
            inb = _tmp0_buffer
            outb = _tmp1_buffer
        else:
            inb = _tmp1_buffer
            outb = _tmp0_buffer

        gs_new = gs // _work_group_size // 2
        if gs_new:
            ws = _work_group_size
            gs = gs_new
        else:
            ws = gs // 2
            gs = 1
        if gs == 1:
            outb = _value_buffer

        function(inb, outb, block = (ws, 1, 1), grid
= (gs, 1), shared = _work_group_size * 4)

        if outb == _value_buffer:
            return

        i += 1
```

The identification of each individual thread is given by the threadIdx value, threadIdx is used to tell each individual thread its identity [13]. This is usually used to determine an index for what values should be processed on the input and output data arrays. Blocks are further executed in abstract batches known as grids, which are best thought of as blocks of blocks [8][9]. As with threads in a block, each block in the grid can be indexed in up to three dimensions with the constant values that are given by blockIdx.x, blockIdx.y, and blockIdx.z.

The grids and blocks are defined [8]-[12]:

```
image_side = len(img_list[0])

global _image_size
_image_size = image_side * image_side

global _work_group_size
_work_group_size =
cuda.Device.get_attribute(pycuda.autoinit.device,
pycuda._driver.device_attribute.MAX_THREADS_PER_BLOC
K)

block_side = int(math.sqrt(_work_group_size))

global _grid_size
_grid_size = _image_size // _work_group_size

half_window_blur = avg_window_size // 2
block_side_blur = block_side - half_window_blur * 2
block_side_3x3 = block_side - 1 * 2
block_side_median = block_side - select_window_size
* 2
grid_side = image_side // block_side

def get_grid_side(bs):
    result = image_side // bs
    if image_side % bs:
        result += 1
    return result

grid_side_blur = get_grid_side(block_side_blur)
grid_side_3x3 = get_grid_side(block_side_3x3)
grid_side_median = get_grid_side(block_side_median)
```

SourceModule actually compiles code into a CUDA module, this is like a Python module, only it contains a collection of compiled CUDA code [12][13].

The whole kernel function then works as follows:

```
module = SourceModule("""
const int imageSide = """ + str(image_side) + """;

const int windowBlur = """ + str(avg_window_size) +
""";
const int halfwindowBlur = """ +
str(half_window_blur) + """;
const int windowMedian = """ +
str(select_window_size * 2 + 1) + """;
const int halfWindowMedian = """ +
str(select_window_size) + """;

const int blockSide = """ + str(block_side) + """;
const int blockSideBlur = """ + str(block_side_blur)
+ """;
const int blockSide3x3 = """ + str(block_side_3x3) +
""";
const int blockSideMedian = """ +
str(block_side_median) + """;


//border functions
__device__ int reflect(int z)
{
    return z < 0 ? -z : (z >= imageSide ? 2 *
imageSide - z - 2 : z);
}

__device__ int clamp(int z)
{
    return z < 0 ? 0 : (z >= imageSide ? imageSide -
1 : z);
}

//index functions
__device__ size_t indexAtR(int x, int y)
{
    return reflect(y) * imageSide + reflect(x);
}

__device__ size_t indexAtC(int x, int y)
{
    return clamp(y) * imageSide + clamp(x);
}

__device__ size_t indexAt(int x, int y)
{
    return y * imageSide + x;
}
```

The following function applies median filter on labeled pixels in order to remove the spikes only on detected noise pixels, it takes input image, binary image of spikes and window size (radius) and returns the filtered image later [10]-[12][14]. For finding white pixels as noise, the following functions are applied:

```
__global__ void blur(const float *imageIn, float
*imageOut)
{
    const int wgx = threadIdx.x - halfwindowBlur;
    const int wgy = threadIdx.y - halfwindowBlur;

    const int x = blockIdx.x * blockSideBlur + wgx;
    const int y = blockIdx.y * blockSideBlur + wgy;

    __shared__ float tmp[blockSide][blockSide];
    const float pixel = imageIn[indexAtR(x, y)];
    tmp[threadIdx.y][threadIdx.x] = pixel;
    __syncthreads();

    if(wgx < 0 || wgx >= blockSideBlur || wgy < 0 ||
wgy >= blockSideBlur)
        return;

    if(x >= imageSide || y >= imageSide)
        return;

    const size_t index = indexAt(x, y);


    float sum = 0;

    for(int i = 0; i < windowBlur; i++) {
        for(int j = 0; j < windowBlur; j++)
            sum += tmp[threadIdx.y - halfwindowBlur
+ i][threadIdx.x - halfwindowBlur + j];
    }

    imageOut[index] = pixel * windowBlur *
windowBlur / sum;
}



__device__ void writeMax(int i, float *tmp)
{
    if(threadIdx.x < i)
        tmp[threadIdx.x] = max(tmp[threadIdx.x],
tmp[threadIdx.x + i]);
}

__global__ void imageMax(const float *imageIn, float
*maxValue)
{
    const size_t gid = blockIdx.x * blockDim.x +
threadIdx.x;

    extern __shared__ float tmp[];
    tmp[threadIdx.x] = max(imageIn[2 * gid],
imageIn[2 * gid + 1]);
    __syncthreads();


    const int halfWorkGroupSize = blockDim.x / 2;

    for(int i = halfWorkGroupSize; i > 32; i >>= 1)
{
        writeMax(i, tmp);
        __syncthreads();
    }

    for(int i = min(halfWorkGroupSize, 32); i > 1; i
>>= 1)
        writeMax(i, tmp);

    if(threadIdx.x == 0)
        maxValue[blockIdx.x] = max(tmp[0], tmp[1]);
}


__device__ void writeMin(int i, float *tmp)
{
    if(threadIdx.x < i)
        tmp[threadIdx.x] = min(tmp[threadIdx.x],
tmp[threadIdx.x + i]);
}

__global__ void imageMin(const float *imageIn, float
*minValue)
{
    const size_t gid = blockIdx.x * blockDim.x +
threadIdx.x;

    extern __shared__ float tmp[];
    tmp[threadIdx.x] = min(imageIn[2 * gid],
imageIn[2 * gid + 1]);
    __syncthreads();


    const int halfWorkGroupSize = blockDim.x / 2;

    for(int i = halfWorkGroupSize; i > 32; i >>= 1)
{
        writeMin(i, tmp);
        __syncthreads();
    }

    for(int i = min(halfWorkGroupSize, 32); i > 1; i
>>= 1)
        writeMin(i, tmp);

    if(threadIdx.x == 0)
        minValue[blockIdx.x] = min(tmp[0], tmp[1]);
}



__device__ float gradient(const float
tmp[blockSide][blockSide], bool dx, bool dy)
{
    return 0
        - tmp[threadIdx.y - 1][threadIdx.x - 1]
        - tmp[threadIdx.y - 1][threadIdx.x    ] * 2
* dy
        + tmp[threadIdx.y - 1][threadIdx.x + 1] *
(dx - dy)
        - tmp[threadIdx.y    ][threadIdx.x - 1] * 2
* dx
        + tmp[threadIdx.y    ][threadIdx.x + 1] * 2
* dx
        + tmp[threadIdx.y + 1][threadIdx.x - 1] *
(dy - dx)
        + tmp[threadIdx.y + 1][threadIdx.x    ] * 2
* dy
        + tmp[threadIdx.y + 1][threadIdx.x + 1]
    ;
}
```

After that, authors have used Sobel operator for detecting edges [1]. The purpose of this operation is to find the regions having sharp change in color within an image; a steep change is indicated by a high value and a shallow change is indicated by a low value. Edge detection has got its significance because it filters out unwanted information and decreases the amount of

data, while still conserving the significant structural characteristics of the image [20]. The Sobel operator calculates the gradient of intensity of a particular image at each pixel and finds the path of the biggest intensification from light contrast pixels to dark contrast pixels; it also calculates the rate of variation in that path [20][21].

Two 3 x 3 kernels are used in the Sobel filter; one changes in the vertical direction and the other one in the horizontal direction. Then for calculating the estimates of the derivatives, both of them are convolved with the original image. For an instance, two images Gx and Gy are defined to contain the vertical and horizontal derivative estimates correspondingly, the computations are described as follows [21]:

$$\boldsymbol{G_y} = \begin{bmatrix} \boldsymbol{+1} & \boldsymbol{+2} & \boldsymbol{+1} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{-1} & \boldsymbol{-2} & \boldsymbol{-1} \end{bmatrix} * \boldsymbol{A}$$

And $$\boldsymbol{G_x} = \begin{bmatrix} \boldsymbol{-1} & \boldsymbol{0} & \boldsymbol{+1} \\ \boldsymbol{-2} & \boldsymbol{0} & \boldsymbol{+2} \\ \boldsymbol{-1} & \boldsymbol{0} & \boldsymbol{+1} \end{bmatrix} * \boldsymbol{A}$$

Where A indicates the original image.

For computing Gx and Gy the suitable kernel window is moved over the input image; then it multiplies the kernel's value with image's each pixel value and then shift the kernel by one pixel to the right. In this way, the kernel is moved over the whole image.

The following code snippet shows how this Sobel operator for this application was implemented in PyCUDA:

```
__global__ void sobel(const float *imageIn, float
*imageOut, float *maxValue)
{
    const int wgx = threadIdx.x - 1;
    const int wgy = threadIdx.y - 1;

    const int x = blockIdx.x * blockSide3x3 + wgx;
    const int y = blockIdx.y * blockSide3x3 + wgy;

    __shared__ float tmp[blockSide][blockSide];
    tmp[threadIdx.y][threadIdx.x] =
min(imageIn[indexAtR(x, y)] / (*maxValue), 1.0);
    __syncthreads();

    if(wgx < 0 || wgx >= blockSide3x3 || wgy < 0 ||
wgy >= blockSide3x3)
        return;

    if(x >= imageSide || y >= imageSide)
        return;

    const size_t index = indexAt(x, y);

    if(x == 0 || x == imageSide - 1 || y == 0 || y
== imageSide - 1) {
        imageOut[index] = 0;
        return;
    }

    const float gradientX = gradient(tmp, 1, 0);
    const float gradientY = gradient(tmp, 0, 1);

    imageOut[index] = sqrt((gradientX * gradientX +
gradientY * gradientY) / 32);
}


__global__ void percentile(const float *imageIn,
float value, int *gLess)
{
    __shared__ int lLess;
    lLess = 0;
    __syncthreads();

    if(imageIn[blockIdx.x * blockDim.x +
threadIdx.x] < value)
        atomicAdd(&lLess, 1);

    __syncthreads();

    if(threadIdx.x == 0)
        atomicAdd(gLess, lLess);
}
```

Again, for finding noises and spikes in an image, some important morphological analysis is done accordingly. Since, the areas having low (valleys in topographical terms) and high (peaks in topographical terms) in an image, can certainly indicate significant morphological characteristics by marking important related image objects [22]. An h-maxima morphological transformation puts down any pixel's intensity with a height equal or smaller than a certain threshold value **h** within a particular intensity area [22]. There can be several local maxima or local minima within an image, but it will have only a one global maxima or minima. An example is shown in the following figure 3:

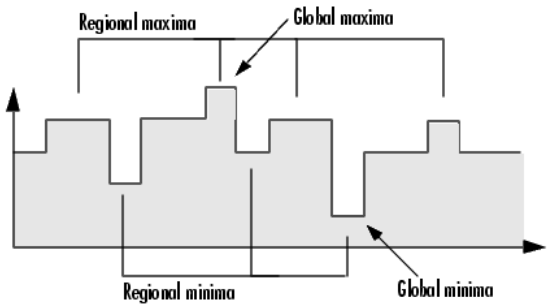


Fig 3: the concept of regional and global Maxima and Minima in an image

First the authors of this work have found out the local maxima with a height of 1 and created the dilation image. The dilation of an image *I* by using a structuring component S_c means the projection to each pixel in the final output image with the highest value which was found in the neighborhood of the pixels. Here the structuring component S_c defines the neighborhood. For a gray image this operation is important since it magnifies the bright features in the image and helps to remove the dark features which are smaller than S_c. So, the dilated image implementation in CUDA according to the authors' specification goes as follows:

```
__device__ bool dilate(const bool
tmp[blockSide][blockSide])
{
    return false
        | tmp[threadIdx.y - 1][threadIdx.x    ]
        | tmp[threadIdx.y    ][threadIdx.x - 1]
        | tmp[threadIdx.y    ][threadIdx.x    ]
        | tmp[threadIdx.y    ][threadIdx.x + 1]
        | tmp[threadIdx.y + 1][threadIdx.x    ]
    ;
}

__global__ void binaryDilation(const float *imageIn,
bool *imageOut, float edgeTop)
{
    const int wgx = threadIdx.x - 1;
    const int wgy = threadIdx.y - 1;

    const int x = blockIdx.x * blockSide3x3 + wgx;
    const int y = blockIdx.y * blockSide3x3 + wgy;

    __shared__ bool tmp[blockSide][blockSide];
    tmp[threadIdx.y][threadIdx.x] =
(imageIn[indexAtR(x, y)] > edgeTop);
    __syncthreads();

    if(wgx < 0 || wgx >= blockSide3x3 || wgy < 0 ||
wgy >= blockSide3x3)
        return;

    if(x >= imageSide || y >= imageSide)
        return;

    imageOut[indexAt(x, y)] = dilate(tmp);
}
```

```
__device__ void write(float *imageOut, size_t index,
unsigned short value, const unsigned short
*brightAverage)
{
    imageOut[index] = min(float(value) /
brightAverage[index], 1.0);
}


__global__ void median(const unsigned short
*imageIn, const bool *noiseMap, const unsigned short
*brightAverage, float *imageOut)
{
    const int wgx = threadIdx.x - halfWindowMedian;
    const int wgy = threadIdx.y - halfWindowMedian;

    const int x = blockIdx.x * blockSideMedian +
wgx;
    const int y = blockIdx.y * blockSideMedian +
wgy;

    __shared__ unsigned short
lTmp[blockSide][blockSide];
    const unsigned short pixel = imageIn[indexAtC(x,
y)];
    lTmp[threadIdx.y][threadIdx.x] = pixel;
    __syncthreads();

    if(wgx < 0 || wgx >= blockSideMedian || wgy < 0
|| wgy >= blockSideMedian)
        return;

    if(x >= imageSide || y >= imageSide)
        return;

    const size_t index = indexAt(x, y);

    if(!noiseMap[index]) {
        write(imageOut, index, pixel,
brightAverage);
        return;
    }

    const int windowSize = windowMedian *
windowMedian;
    unsigned short pTmp[windowSize];

    for(int i = 0; i < windowMedian; i++) {
        for(int j = 0; j < windowMedian; j++)
            pTmp[i * windowMedian + j] = lTmp[wgy +
i][wgx + j];
    }


    for(int j = 0; j < windowSize - 1; j++) {
        unsigned short m = pTmp[j];
        int mi = j;

        for(int i = j + 1; i < windowSize; i++) {
            if(pTmp[i] > m) {
                m = pTmp[i];
                mi = i;
            }
        }

        pTmp[mi] = pTmp[j];
        pTmp[j] = m;
    }

    write(imageOut, index, pTmp[windowSize / 2],
brightAverage);
}
""")
```

Finally, the following section copies input to GPU and then copies output to host from GPU:

```
#copy input to gpu
cuda.memcpy_htod(image_float0_buffer,
img.astype(np.float32))

#actual image processing
blur(image_float0_buffer, image_float1_buffer, block
= (block_side, block_side, 1), grid =
(grid_side_blur, grid_side_blur))

find_value(_image_max, image_float1_buffer)

sobel(image_float1_buffer, image_float0_buffer,
_value_buffer, block = (block_side, block_side, 1),
grid = (grid_side_3x3, grid_side_3x3))

edge_top = percentile(image_float0_buffer, 95)

binary_dilation(image_float0_buffer,
image_bool_buffer, edge_top, block = (block_side,
block_side, 1), grid = (grid_side_3x3,
grid_side_3x3))

cuda.memcpy_htod(image_ushort0_buffer, img)

cuda.memcpy_htod(image_ushort1_buffer,
bright_average)

median(image_ushort0_buffer, image_bool_buffer,
image_ushort1_buffer, image_float0_buffer, block =
(block_side, block_side, 1), grid =
(grid_side_median, grid_side_median))


#copy output from gpu
img_corrected =
np.empty_like(img).astype(np.float32)
cuda.memcpy_dtoh(img_corrected, image_float0_buffer)

img_noise_map = np.empty_like(img).astype(np.bool)
cuda.memcpy_dtoh(img_noise_map, image_bool_buffer)


img_corrected =
inpaint.inpaint_biharmonic(img_corrected, ipmask,
multichannel=False)
io.imsave(out_bf+'/'+fname+'{0:04}'.format(i)+fmt_ou
t, img_corrected)
```

In this way, the PyCUDA execution of the respective application was implemented.

5. Experimental Evaluation and Results:

I tried to test this image processing application in Discus-cloud. I was interested to see how much performance benefits I can get if GPUs are used alongside with CPUs for the same application. As the overall completion time of running the application indicates whether GPUs are performing better compared to CPUs or not, the performance metric that I have mainly considered in my project is the overall completion time. I have considered these parameters as primary factors - types of CPUs, types of GPUs and number of GPUs.

The CPU types for my tests are the CPU model of Discus-cloud, which is Intel(R) Xeon(R) CPU E5-2620 v3 (Discus-cloud). For GPUs, Discus-cloud has GRID k1 (4 GPUs). For accessing the data, the data are fetched from NFS.

5.1 Experimental Setup:

I ran all these experiments on Discus-cloud which has Intel(R) Xeon(R) CPU E5-2620 v3 clocked at 2.40GHz. The processor has 6 cores per socket, 2 threads for each core and total 2 sockets which means it has 6*2*2=24 computing units. The GPU version of Discus-cloud is NVIDIA GRID k1 (4GPUs) which has total 768 NVIDIA CUDA® cores (192 per GPU), 16 GB of memory (4 GB per GPU) and memory I/O bandwidth of 128-bit.

For this image processing application, I used python 3, OpenCV, PyCUDA and NVIDIA CUDA toolkit- 8.0 with driver version 367 by using a python virtual environment. For profiling the CUDA parts I used nvprof() and for profiling the GPU memory usage, I wrote a bash script that periodically ran nvidia-smi command for 2 seconds of interval.

5.2 Results:

I have a dataset of 366 images. I have run my code for both CPU and GPU on the whole dataset, the data were equally distributed among the four GPUs. I ran all these experiments while the dataset was accessed from NFS. It took 26 minutes to perform the processing in CPU and 17 minutes in GPUs, which means it performed more than 1.5x times faster than CPU. The runtime result among their time consumption is shown as bar-chart comparison in Figure 4.

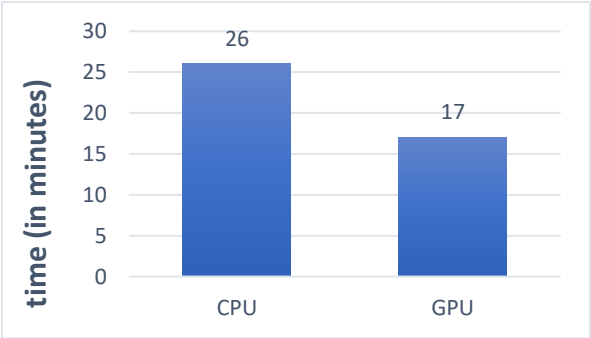
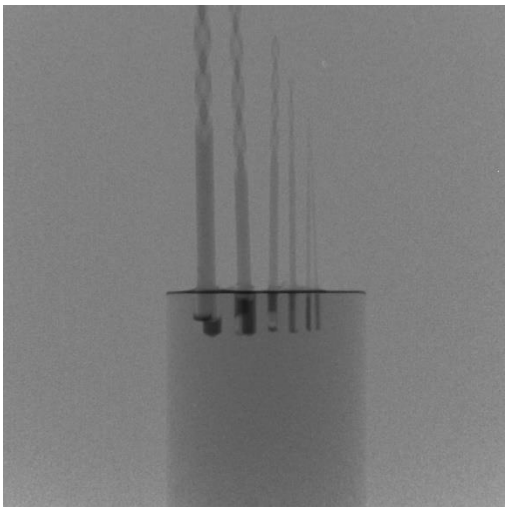


Fig 4: Time comparison (in minutes) of the image processing application in CPU vs. GPUs

And the output images looked like as shown in figure 5:

(a)



(b)

Fig 5: (a) Raw image (contains spike noise) (b) after removing spike noise and bright field correction

After profiling the application, in terms of memory usage, I found the maximum amount of GPU memory consumed by the application is 62 MB out 4096 MB for each GPU. That means for each GPU, only around 1.514% of the total memory was used for the application. As it is seen the GPU memories were not utilized that much, according to my opinion the probable reason is that there is enough space for computing more images at once in a single GPU, so processing only around 366/4 ≈ 91 images per GPU core is comparatively a very small data load for GPUs.

6. Conclusion:

In this report, I tried to show how an image processing application can take advantage of mulit-GPU parallelism. I found that, splitting the data among four GPU cores performed more than 1.5x better than processing in CPU, in terms of task completion time. From profiling the memory usage, it was observed for each GPU, only 1.5% of the GPU memory is utilized for this image processing application which means GPUs are able to process more data at a time.

Acknowledgements

1. Robust and User Friendly 3D Re-Construction of Neutron Tomographic Images, Hao Song ; Mark Eramian ; Emil Hallin ; Blanche Leyeza ; Paul G. Arnison ; Ronald Rogge; 2018 IEEE Winter Conference on Applications of Computer Vision (WACV); Year: 2018; Pages: 930 – 938.
2. https://devblogs.nvidia.com/even-easier-introduction-cuda/
3. https://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf
4. https://www.tutorialspoint.com/cuda/index.htm
5. https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/
6. https://docs.computecanada.ca/wiki/CUDA_tutorial
7. https://jhui.github.io/2017/03/06/CUDA/
8. http://supercomputingblog.com/cuda-tutorials/
9. https://developer.nvidia.com/pycuda
10. https://weeraman.com/put-that-gpu-to-good-use-with-python-e5a437168c01
11. https://nyu-cds.github.io/python-numba/05-cuda/
12. https://documen.tician.de/pycuda/tutorial.html
13. https://medium.com/3blades-blog/an-introduction-to-gpu-programming-with-python-637818be6f7d
14. https://linuxhint.com/gpu-programming-python/
15. https://docs.anaconda.com/numbapro/CUDAJit/

16. http://www.numpy.org/
17. https://scikit-image.org/docs/dev/auto_examples/filters/plot_inpaint.html
18. https://en.wikipedia.org/wiki/Median_filter
19. https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
20. https://medium.com/datadriveninvestor/understanding-edge-detection-sobel-operator-2aada303b900.
21. https://www.cs.auckland.ac.nz/compsci373s1c/PatricesLectures/Edge%20detection-Sobel_2up.pdf
22. https://edoras.sdsu.edu/doc/matlab/toolbox/images/morph14.html