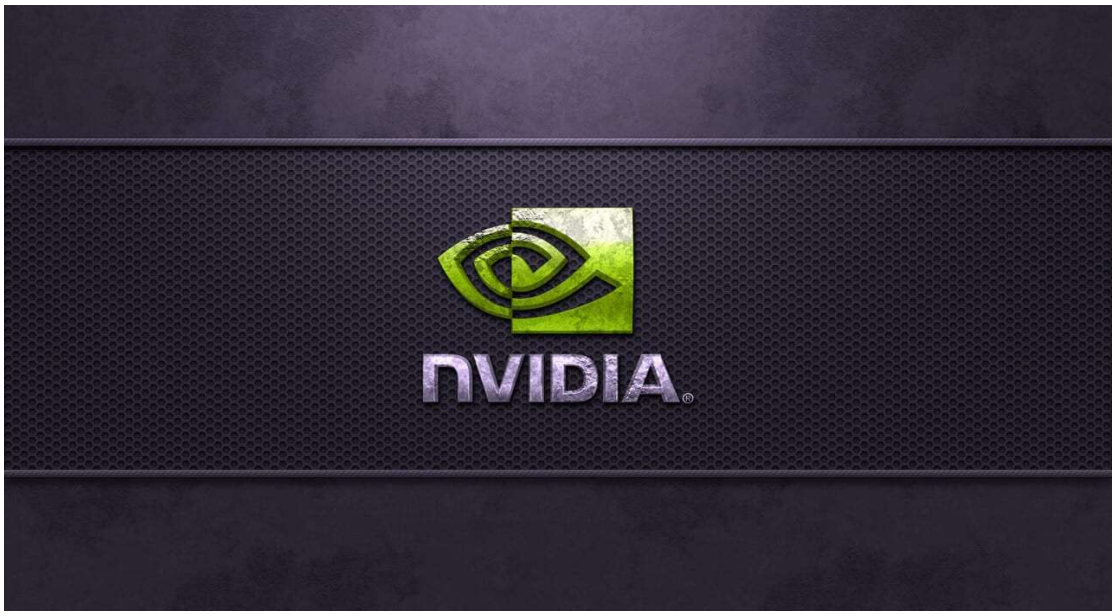
PROJECT 2 CUDA



NAMES: NTONES SAVVAS, PETROUDIS NIKOLAOS
PROFESSOR: IRAKLIS SPILIOTIS
SUBJECT: PARALLEL PROCESSING TECHNOLOGY
TOPIC: BINARIZATION OF GREY IMAGES USING OTSU METHOD
LABORATORY OF COMPUTER ARCHTECTURE AND HIGH-PERFORMANCE SYSTEMS
DEPARTURE OF ELECTRICAL AND COMPUTER ENGINEERING
DEMOCRETUS UNIVERSITY OF THRACE
ACADEMIC SEASON: 2022 – 2023

INTRODUCTION

A binary image is created by dividing a gray-scale image into two levels, usually called area 0 and area 1 [2]. Thresholding is an important technique of computer vision and image processing, by which a target-object is separated from the background image [1]. Using Otsu method, a threshold is found automatically in order image binarization to be achieved [2]. The optimal threshold is chosen by the classification criterion to separate the two categories (foreground and background. Specifically, threshold is determined by minimizing intra-class intensity or by maximizing inter-class variance, which is defined as a weighted variance of average intensity values of the two categories. [1].

As a result, image is separated into two areas, the bright area T0 and the dark area T1. T0 area contains intensity levels from 0 to t, while T1 region contains intensity levels from t to l, where t is the threshold and l is the maximum intensity level (i.e. 256). T0 and T1 can correspond to object and background or vice versa (the bright area does not always correspond to the object) [4].

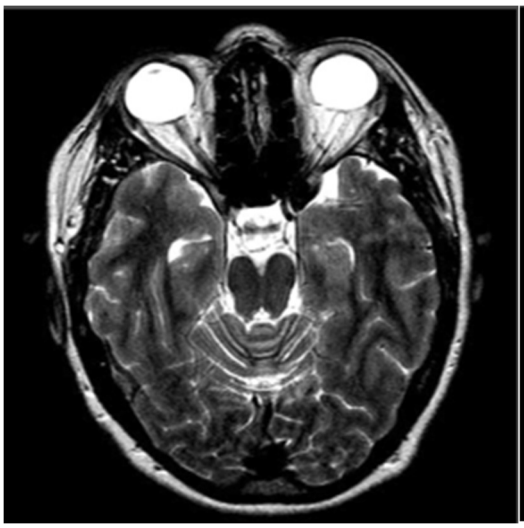


Figure 1- mri 4096x4096 grey

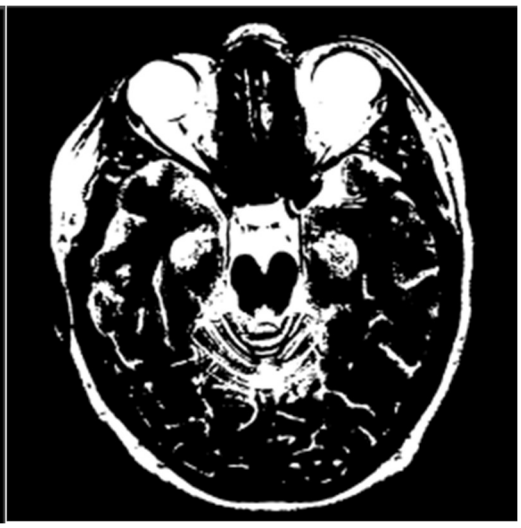


Figure 2- mri 4096x4096 cuda2

Figure 1- drone 16384x16384 grey

Figure 2- drone 16384x16384 cuda2

PSEUDOCODE

STEP 1: Histogram computation of grey-scale image.

STEP 2 (Repeated) : Foreground and background variance computation for one threshold value each time.

i) Computation of foreground and background pixels' weight.
ii) Computation of foreground and background pixels' average value.
iii) Computation of foreground and background pixels' variance.

STEP 3: Inter-class variance computation and the optimal threshold is determined by the maximum value of inter-class variance.

STEP 4: Intensity values less than the threshold value are determined to new intensity value 0, otherwise 255.

```
void copy_in_2_out_img (length, width, inimg, outimg)
    unsigned long length, width;
    unsigned char inimg[length][width], outimg[length][width];
{
    int total=0;
    int top=256;
    int sumB=0;
    int wB=0;
    int maximum=0;
    int sum1=0;

    int hist[256];
    int i,j,temp;
    double start,end;
    start = omp_get_wtime();

    for (i=0;i<=255;i++)
        hist[i] = 0;

    for(i=0;i<length;i++)
    {
    for(j=0;j<width;j++)
    {
      temp = inimg[i][j];
      hist[temp] += 1;
    }
    }


for(i=0;i<top;i++)
{
    sum1=sum1+i*hist[i];
    total=total+hist[i];
}
```

Image 3 – Serial code 1

```
int wF,mF;
int level,val;
for(i=1;i<=top;i++)
{   wF=total-wB;
    if(wB>0 && wF>0)
{
    mF=(sum1-sumB)/wF;
    val=wB*wF*((sumB/wB)-mF)*((sumB/wB)-mF);
    if(val>=maximum)
{
    level=i;
    maximum=val;
}
}

    wB=wB+hist[i];
    sumB=sumB+(i-1)*hist[i];

}


    for(i=0;i<length;i++)
  {
    for(j=0;j<width;j++)
    {
        if(inimg[i][j]<level)
        {
        outimg[i][j]=0;
        }
        else
        {
            outimg[i][j]=255;
        }
    }
}
```

Image 4 – Serial code 2

The time complexity of this algorithm is O(N*M + 2*K), where N,M are the height and width of image respectively and K is the maximum intensity value of grey-scale image.

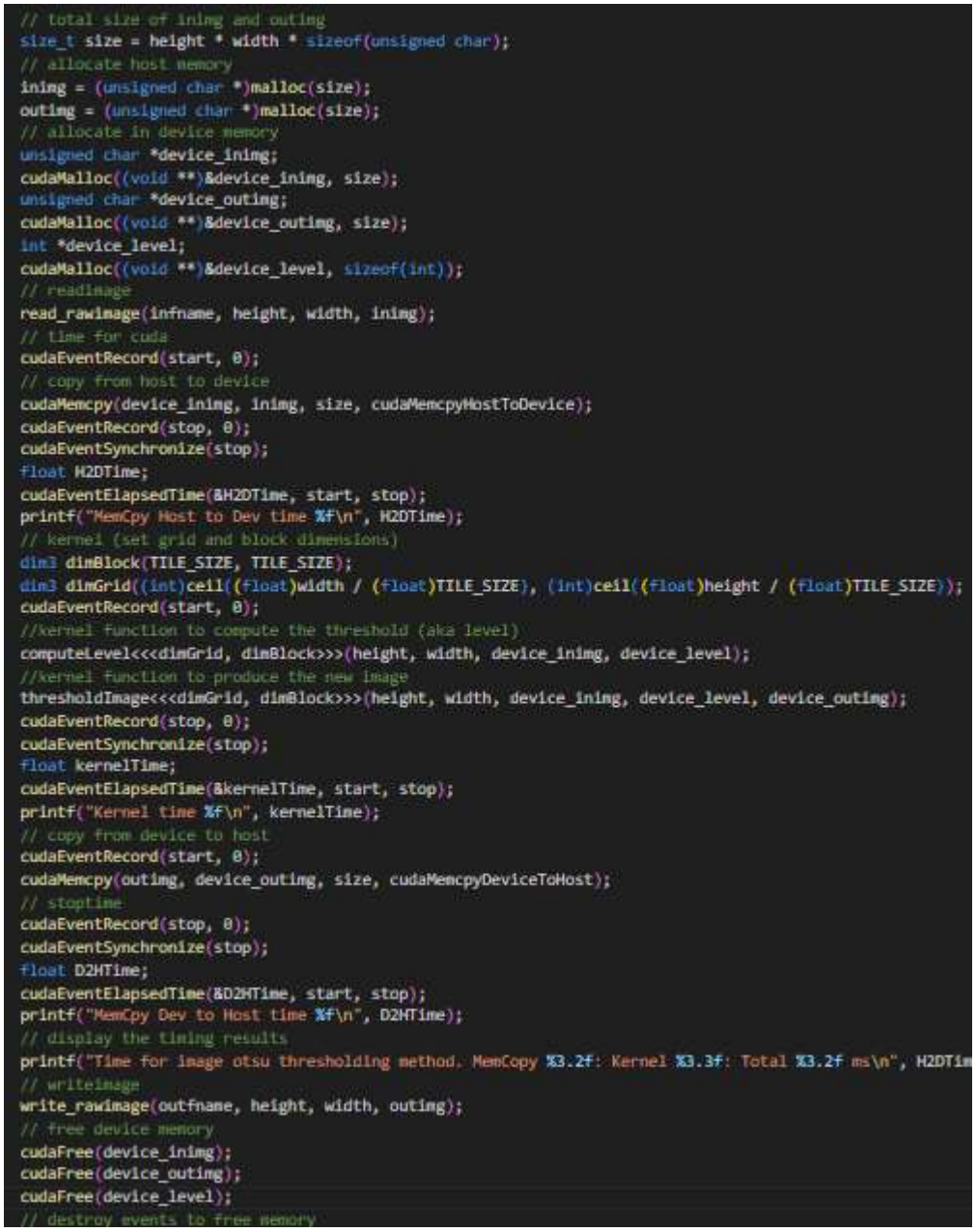
Code's parallelization

```
__global__ void computeLevel(unsigned long height, unsigned long width, unsigned char *device_inimg, int *device_level)
{
    __shared__ int hist[256];
    __shared__ int sum1;
    __shared__ int sumB;
    __shared__ int total;
    __shared__ int wB;
    if (threadIdx.x < 256)
        hist[threadIdx.x] = 0;
    __syncthreads();

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int temp, k, wF, maximum, level, mF, val;
    if (i < height && j < width)
    {
        temp = device_inimg[i * width + j];
        atomicAdd(&hist[temp], 1);
    }
    __syncthreads();
    if (threadIdx.x == 0 && threadIdx.y == 0)
    {
        total = 0;
        sum1 = 0;
        for (k = 0; k < 256; k++)
        {
            total += hist[k];
            sum1 += k * hist[k];
        }
        __syncthreads();
        wB = 0;
        sumB = 0;
        maximum = 0;
        level = 0;

        for (k = 0; k < 256; k++)
        {
            wB = wB + hist[k];
            sumB = sumB + (k-1)*hist[k];
            wF = total - wB;
            if (wB > 0 && wF > 0)
            {
                mF = (sum1 - sumB) / wF;
                val = wB * wF * ((sumB / wB) - mF) * ((sumB / wB) - mF);
                if (val >= maximum)
                {
                    level = k;
                    maximum = val;
                }
            }
        }
        __syncthreads();
        *device_level = level;
    }
}
```

Figure 3- kernel function 1

```
__global__ void thresholdImage(unsigned long height, unsigned long width, unsigned char *device_inimg, int *device_level, unsigned char *device_outimg)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int level = *device_level;
    if (i < height && j < width)
    {
        if (device_inimg[i * width + j] < level)
        {
            device_outimg[i * width + j] = 0;
        }
        else
        {
            device_outimg[i * width + j] = 255;
        }
    }
}
```

Figure 4- kernel function 2



```
// total size of inimg and outimg
size_t size = height * width * sizeof(unsigned char);
// allocate host memory
inimg = (unsigned char *)malloc(size);
outimg = (unsigned char *)malloc(size);
// allocate in device memory
unsigned char *device_inimg;
cudaMalloc((void **)&device_inimg, size);
unsigned char *device_outimg;
cudaMalloc((void **)&device_outimg, size);
int *device_level;
cudaMalloc((void **)&device_level, sizeof(int));
// readimage
read_rawimage(infname, height, width, inimg);
// time for cuda
cudaEventRecord(start, 0);
// copy from host to device
cudaMemcpy(device_inimg, inimg, size, cudaMemcpyHostToDevice);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float H2DTime;
cudaEventElapsedTime(&H2DTime, start, stop);
printf("MemCpy Host to Dev time %f\n", H2DTime);
// kernel (set grid and block dimensions)
dim3 dimBlock(TILE_SIZE, TILE_SIZE);
dim3 dimGrid((int)ceil((float)width / (float)TILE_SIZE), (int)ceil((float)height / (float)TILE_SIZE));
cudaEventRecord(start, 0);
//kernel function to compute the threshold (aka level)
computeLevel<<<dimGrid, dimBlock>>>(height, width, device_inimg, device_level);
//kernel function to produce the new image
thresholdImage<<<dimGrid, dimBlock>>>(height, width, device_inimg, device_level, device_outimg);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float kernelTime;
cudaEventElapsedTime(&kernelTime, start, stop);
printf("Kernel time %f\n", kernelTime);
// copy from device to host
cudaEventRecord(start, 0);
cudaMemcpy(outimg, device_outimg, size, cudaMemcpyDeviceToHost);
// stoptime
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float D2HTime;
cudaEventElapsedTime(&D2HTime, start, stop);
printf("MemCpy Dev to Host time %f\n", D2HTime);
// display the timing results
printf("Time for image otsu thresholding method. MemCopy %3.2f: Kernel %3.3f: Total %3.2f ms\n", H2DTim
// writeimage
write_rawimage(outfname, height, width, outimg);
// free device memory
cudaFree(device_inimg);
cudaFree(device_outimg);
cudaFree(device_level);
// destroy events to free memory
```

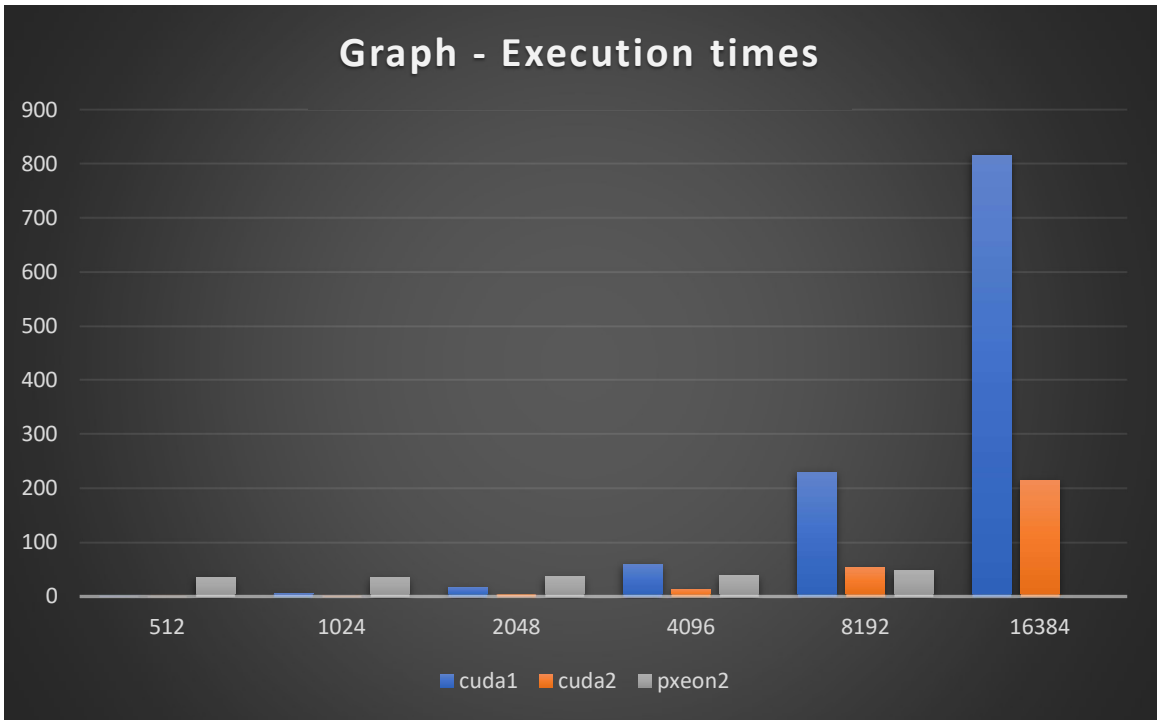
Figure 5- main function

Metrics

Metrics for different image sizes are displayed below. These were calculated in cuda1,where GPU Nvidia GTX950 can be found and cuda2, where GPU RTX3080 can be found. Metrics for different images of pxeon2 are also displayed. Pxeon 2 is a machine which contains 64 processors.

Images	@cuda1			@cuda2		
	MemCopy(ms)	Kernel(ms)	Total(ms)	MemCopy(ms)	Kernel(ms)	Total(ms)
512 x 512	0.29	0.539	**0.83**	0.3	0.068	**0.37**
1Kx1K	0.93	4.244	**5.17**	1.04	0.204	**1.24**
2Kx2K	3.30	14.503	**17.8**	3.39	0.518	**3.91**
4Kx4K	12.30	48.494	**60.79**	12.16	1.782	**13.94**
8Kx8K	47.83	182.486	**230.32**	48.05	6.804	**54.86**
16K x 16K	189.27	627.461	**816.73**	189.07	25.298	**214.37**

Images	@pxeon2 OpenMP64cores (ms)
512 x 512	**35.87**
1Kx1K	**35.89**
2Kx2K	**36.96**
4Kx4K	**38.93**
8Kx8K	**47.87**



Conclusion

Parallel code of binarization of grey-scale image using Otsu method is faster for small images(2048x2048 resolution and below) with cuda compared to OpenMP. Then we can clearly see that OpenMP code is faster because the MemCopy task is executed slowly.

Bibliography

1)Nobuyuki Otsu (1979). "A threshold selection method from gray-level histograms". *IEEE Trans. Sys. Man. Cyber*. **9** (1): 62–66. https://ieeexplore.ieee.org/document/4310076

2) Sunil L. Bangare, Amruta Dubal, Pallavi S. Bangare, Dr. S.T. Patil, Reviewing Otsu's Method For Image Thresholding, International Journal of Applied Engineering Research, https://dx.doi.org/10.37622/IJAER/10.9.2015.21777-21783

3) An introduction to parallel programming, Peter S. Pacheco

4) Jamileh Yousefi, Image Binarization using Otsu Thresholding Algorithm, http://dx.doi.org/10.13140/RG.2.1.4758.9284