Exp:9 **AI and Vision Systems Lab**

Parallel Programming Using CUDA C

Aim:
To study parallel programming concepts using CUDA and understand the difference between GPU and CPU processing.

Software/ Packages Used:

1. Google Colaboratory
2. Libraries used:
   - Opencv – python
   - Numpy
   - Matplotlib
   - Tensorflow

Programs:

Installation:
Install CUDA Version 9:

https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-re poubuntu1604-9-2-local_9.2.88-1_amd64 -O cuda-repo-ubuntu1604-9-2-local_9. 2.88-1_amd64.deb

CUDA Version 10.2:

https://colab.research.google.com/drive/1CC5iCU6QSwEBaUqRgVf__yAQ3nVEHrCZ?usp=sharing#scrollTo=Cxj657pT4dRE

Exercise:
CUDA Program:

https://colab.research.google.com/github/ShimaaElabd/CUDA-GPU-ContrastEnhancement/blob/master/CUDA_GPU.ipynb

CPU VS GPU:
https://colab.research.google.com/drive/1Daur4gtaYjmb4XxBFPr7NnFSXtCM8T#scrollTo=eeQ5HzW7FKod

Coding:
Installation:

```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk &#39;{print $2}&#39; | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
Install CUDA Version 9:
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-re
po-ubuntu1604-9-2-local_9.2.88-1_amd64 -O cuda-repo-ubuntu1604-9-2-local_9. 2.88-
1_amd64.deb
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
!apt-get update
!apt-get install cuda-9.2
```

Check the Version of CUDA by : running the command below to get the following output :

```
!export PATH=/usr/local/cuda/bin${PATH:+:${PATH}}
!export LD_LIBRARY_PATH=/usr/local/cuda/lib64\${LD_LIBRARY_PATH:+:${
LD_LIBRARY_PATH}}
!/usr/local/cuda/bin/nvcc --version
```

Execute the given command to install a small extension to run nvcc from Notebook cells:

```
!git config --global url.&quot;https://github.com/&quot;.insteadOf git://github.com/
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

Load the extension using this code:

```
%load_ext nvcc_plugin
```

CUDA Program – 1

```
%%cu
#include &lt;stdio.h&gt; #include &lt;stdlib.h&gt;
global void add(int *a, int *b, int *c) {
*c = *a + *b;}
int main() { int a, b, c;
// host copies of variables a, b &amp; c int *d_a, *d_b, *d_c;
// device copies of variables a, b &amp; c int size = sizeof(int);
// Allocate space for device copies of a, b, c cudaMalloc((void **)&amp;d_a, size);
cudaMalloc((void **)&amp;d_b, size); cudaMalloc((void **)&amp;d_c, size);
// Setup input values c = 0;
a = 3;
b = 5;
// Copy inputs to device
cudaMemcpy(d_a, &amp;a, size, cudaMemcpyHostToDevice); cudaMemcpy(d_b, &amp;b, size,
cudaMemcpyHostToDevice);
// Launch add() kernel on GPU add&lt;&lt;&lt;1,1&gt;&gt;&gt;(d_a, d_b, d_c);
// Copy result back to host
cudaError err = cudaMemcpy(&amp;c, d_c, size, cudaMemcpyDeviceToHost);
if(err!=cudaSuccess) {
printf(&quot;CUDA error copying to Host: %s\n&quot;, cudaGetErrorString(err));}
printf(&quot;result is %d\n&quot;,c);
// Cleanup cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); return 0;}
```

CUDA Program – 2

```
%%cu
#include &lt;stdio.h&gt; #define N 64
inline cudaError_t checkCudaErr(cudaError_t err, const char* msg) { if (err !=
cudaSuccess) {
fprintf(stderr, &quot;CUDA Runtime error at %s: %s\n&quot;, msg, cudaGetErrorString(err)
);}return err;}
global void matrixMulGPU( int * a, int * b, int * c ){
/** Build out this kernel.*/
int row = threadIdx.y + blockIdx.y * blockDim.y; int col = threadIdx.x + blockIdx.x *
blockDim.x;
int val = 0;
if (row &lt; N &amp;&amp; col &lt; N) { for (int i = 0; i &lt; N; ++i) {
val += a[row * N + i] * b[i * N + col];}
```

```
c[row * N + col] = val;}}
/** This CPU function already works, and will run to create a solution matrix
* against which to verify your work building out the matrixMulGPU kernel.*/
void matrixMulCPU( int * a, int * b, int * c )
{int val = 0;
for( int row = 0; row &lt; N; ++row ) for( int col = 0; col &lt; N; ++col )
{val = 0;
for ( int k = 0; k &lt; N; ++k )
val += a[row * N + k] * b[k * N + col]; c[row * N + col] = val;}}
int main()
{
int *a, *b, *c_cpu, *c_gpu; // Allocate a solution matrix for both the CPU and the GPU
operations
int size = N * N * sizeof (int); // Number of bytes of an N x N matrix
// Allocate memory cudaMallocManaged (&amp;a, size); cudaMallocManaged (&amp;b, size);
cudaMallocManaged (&amp;c_cpu, size); cudaMallocManaged (&amp;c_gpu, size);
// Initialize memory; create 2D matrices for( int row = 0; row &lt; N; ++row ) for( int col =
0; col &lt; N; ++col )
{a[row*N + col] = row; b[row*N + col] = col+2; c_cpu[row*N + col] = 0; c_gpu[row*N + col]
= 0;}
/** Assign `threads_per_block` and `number_of_blocks` 2D values
* that can be used in matrixMulGPU above.*/
dim3 threads_per_block(32, 32, 1);
dim3 number_of_blocks(N / threads_per_block.x + 1, N / threads_per_block.y + 1, 1);
matrixMulGPU &lt;&lt;&lt; number_of_blocks, threads_per_block &gt;&gt;&gt; ( a, b, c_gpu );
checkCudaErr(cudaDeviceSynchronize(), &quot;Syncronization&quot;);
checkCudaErr(cudaGetLastError(), &quot;GPU&quot;);
// Call the CPU version to check our work matrixMulCPU( a, b, c_cpu );
// Compare the two answers to make sure they are equal bool error = false;
for( int row = 0; row &lt; N &amp;&amp; !error; ++row ) for( int col = 0; col &lt; N &amp;&amp;
!error; ++col )
if (c_cpu[row * N + col] != c_gpu[row * N + col])
{
printf(&quot;FOUND ERROR at c[%d][%d]\n&quot;, row, col); error = true;
break;}
if (!error) printf(&quot;Success!\n&quot;);
// Free all our allocated memory cudaFree(a); cudaFree(b);
cudaFree( c_cpu ); cudaFree( c_gpu );}
```

<u>CUDA Program – 3</u>

```
%%cu #include&lt;stdio.h&gt; #include&lt;cuda.h&gt;
int main()
{
cudaDeviceProp p; int device_id;
int major; int minor;
cudaGetDevice(&amp;device_id); cudaGetDeviceProperties(&amp;p,device_id);
major=p.major; minor=p.minor;
printf(&quot;Name of GPU on your system is %s\n&quot;,p.name);
printf(&quot;\n Compute Capability of a current GPU on your system is %d.%d&quot;,majo
r,minor);
return 0;
}
```

CUDA Program – 4

```
%%cu #include&lt;stdio.h&gt; #include&lt;cuda.h&gt;
#define row1 2 /* Number of rows of first matrix */ #define col1 3 /* Number of
columns of first matrix */ #define row2 3 /* Number of rows of second matrix */ #define
col2 2 /* Number of columns of second matrix */
global void matproductsharedmemory(int *l,int *m, int *n)
{
int x=blockIdx.x; int y=blockIdx.y;
shared int p[col1];
int i;
int k=threadIdx.x; n[col2*y+x]=0; p[k]=l[col1*y+k]*m[col2*k+x];
syncthreads();
for(i=0;i&lt;col1;i++) n[col2*y+x]=n[col2*y+x]+p[i];}
int main()
{int a[row1][col1]; int b[row2][col2]; int c[row1][col2]; int *d,*e,*f;
int i,j;
a[0][0]=2;
a[0][1]=6;
a[0][2]=2;
a[1][0]=4;
a[1][1]=7;
a[1][2]=3;
b[0][0]=2;
b[0][1]=5;
b[1][0]=7;
b[1][1]=1;
b[2][0]=8;
b[2][1]=5;
cudaMalloc((void **)&amp;d,row1*col1*sizeof(int)); cudaMalloc((void
**)&amp;e,row2*col2*sizeof(int)); cudaMalloc((void **)&amp;f,row1*col2*sizeof(int));
cudaMemcpy(d,a,row1*col1*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(e,b,row2*col2*sizeof(int),cudaMemcpyHostToDevice);
dim3 grid(col2,row1);
/* Here we are defining two dimensional Grid(collection of blocks) structure. Synt ax is
dim3 grid(no. of columns,no. of rows) */
matproductsharedmemory&lt;&lt;&lt;grid,col1&gt;&gt;&gt;(d,e,f);
cudaMemcpy(c,f,row1*col2*sizeof(int),cudaMemcpyDeviceToHost);
printf(&quot;\n Product of two matrices:\n &quot;); for(i=0;i&lt;row1;i++)
{for(j=0;j&lt;col2;j++)
{printf(&quot;%d\t&quot;,c[i][j]);}
printf(&quot;\n&quot;);}
cudaFree(d); cudaFree(e); cudaFree(f);
return 0;}
```

CUDA Program – 5

```
%%cu #include&lt;stdio.h&gt; #include&lt;cuda.h&gt;
constant int d[5];
global void add(int *c)
{int id=threadIdx.x;
c[id]=c[id]+d[id];}
int main()
```

```
{int a[5];
int b[5]={1,2,3,4,5};
int *c; int i;
a[0]=1;a[1]=8;a[2]=9;a[3]=6;a[4]=3;
cudaMalloc((void **)&amp;c,5*sizeof(int));
cudaMemcpy(c,a,5*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(d,b,5*sizeof(int)); /*copying contents of array b to cons
tant array d */ add&lt;&lt;&lt;1,5&gt;&gt;&gt;(c);
cudaMemcpy(a,c,5*sizeof(int),cudaMemcpyDeviceToHost);
printf(&quot;Elements of your array after addition with constant array {1,2,3,4,5} :\n&quot;);
for(i=0;i&lt;5;i++)
{printf(&quot;%d\t&quot;,a[i]);}
cudaFree(c); cudaFree(d);}
```

2. CPU VS GPU:

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from google.colab.patches import cv2_imshow # Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)
# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
# Scale images to the [0, 1] range x_train = x_train.astype(&quot;float32&quot;) / 255 x_test =
x_test.astype(&quot;float32&quot;) / 255
# Make sure images have shape (28, 28, 1) x_train = np.expand_dims(x_train, -1) x_test =
np.expand_dims(x_test, -1) print(&quot;x_train shape:&quot;, x_train.shape)
print(x_train.shape[0],
&quot;train samples&quot;) print(x_test.shape[0], &quot;test samples&quot;)
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
model = keras.Sequential( [
keras.Input(shape=input_shape),
layers.Conv2D(32, kernel_size=(3, 3), activation=&quot;relu&quot;),
layers.MaxPooling2D(pool_size=(2, 2)), layers.Conv2D(64, kernel_size=(3, 3),
activation=&quot;relu&quot;), layers.MaxPooling2D(pool_size=(2, 2)),
layers.Flatten(), layers.Dropout(0.5),
layers.Dense(num_classes, activation=&quot;softmax&quot;),])
model.summary()
import datetime print(&quot;Training the Model &quot;)
start_time=datetime.datetime.now() print(&quot;Training started at:
{}&quot;.format(start_time))
batch_size = 128
epochs = 15
model.compile(loss=&quot;categorical_crossentropy&quot;, optimizer=&quot;adam&quot;,
metrics=[&quot;accuracy&quot;])
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
end_time= datetime.datetime.now() print(&quot;Training ended at: {}&quot;.format(end_time))
duration = end_time - start_time print(&quot;Training Duration: {}&quot;.format(duration))
score = model.evaluate(x_test, y_test, verbose=0) print(&quot;Test loss:&quot;, score[0])
```

Hello World Program:

```
!nvcc --version
!pip install pycuda
```

```
!pip install scikit-image
%%writefile hello.cu
#include &quot;stdio.h&quot;
__global__ void hello(void)
{printf(&quot;GPU: Hello!\n&quot;);}
int main(int argc,char **argv)
{printf(&quot;CPU: Hello!\n&quot;);
hello&lt;&lt;&lt;1,10&gt;&gt;&gt;();
cudaDeviceReset();
return 0;}
```

<u>Output :</u>

```
!./hello
CPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
GPU: Hello!
```

Image processing programs:

```
!pip install pycuda
!pip install scikit-image
```

<u>Program 1</u>

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
from skimage.color import rgb2gray
from scipy.ndimage import gaussian_filter
image_url = "/content/sunset-lake.jpg" # Update the image URL to point to the local image file
image = io.imread(image_url)
gray_image = rgb2gray(image)
sigma = 2.0
blur_cpu = gaussian_filter(gray_image, sigma=sigma)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(gray_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(blur_cpu, cmap='gray')
plt.title('Blurred Image (CPU)')
plt.axis('off')
plt.show()
```

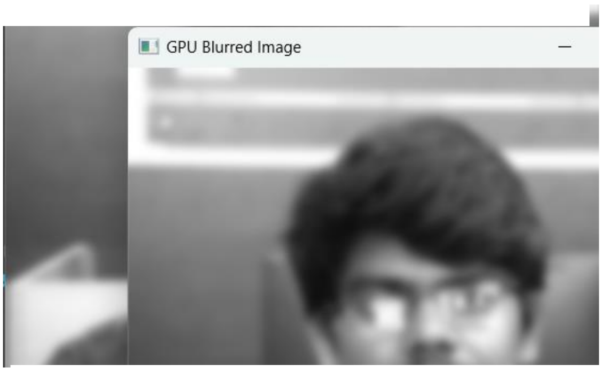
Program 2

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
from skimage.color import rgb2gray
from scipy.ndimage import gaussian_filter
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
image_url = "/content/sunset-lake.jpg"
image = io.imread(image_url)
gray_image = rgb2gray(image)
gray_image_gpu = gpuarray.to_gpu(gray_image.astype(np.float32))
sigma = 2.0
blur_cpu = gaussian_filter(gray_image, sigma=sigma)
blur_gpu = gaussian_filter(gray_image_gpu.get(), sigma=sigma)
plt.figure(figsize=(12, 6))
plt.subplot(1, 3, 1)
plt.imshow(image)
plt.title('Original Image')
plt.axis('off')
plt.subplot(1, 3, 2)
plt.imshow(blur_cpu, cmap='gray')
plt.title('Blurred Image (CPU)')
plt.axis('off')
plt.subplot(1, 3, 3)
plt.imshow(blur_gpu, cmap='gray')
plt.title('Blurred Image (GPU)')
plt.axis('off')
plt.show()
```

Outputs of the scripts

ORIGINAL IMAGE

CPU BLURRED



GPU BLURRED

EDGE-DETECTED IMAGE-CUDA

Department of RAE			
Criteria	Excellent (75% - 100%)	Good (50 – 75%)	Poor (<50%)
Preparation (30)			
Performance (30)			
Evaluation (20)			
Report (20)			
Sign:		Total (100)	

Result:

Thus the parallel programming concepts and the difference between CPU and GPU programming were studied.