Experiment 9

Parallel Programming Using CUDA

# 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:

**Coding:**

# Installation:

!apt-get --purge remove cuda nvidia\* libnvidia-\*

!dpkg -l | grep cuda- | awk '{print $2}' | 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-repo- 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

!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."https://github.com/".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 <stdio.h> #include <stdlib.h> global void add(int \*a, int \*b, int \*c) {

\*c = \*a + \*b;

}

int main() { int a, b, c;

// host copies of variables a, b & c int \*d\_a, \*d\_b, \*d\_c;

// device copies of variables a, b & c int size = sizeof(int);

// Allocate space for device copies of a, b, c cudaMalloc((void \*\*)&d\_a, size); cudaMalloc((void \*\*)&d\_b, size); cudaMalloc((void \*\*)&d\_c, size);

// Setup input values c = 0; a = 3;

b = 5;

// Copy inputs to device

cudaMemcpy(d\_a, &a, size, cudaMemcpyHostToDevice); cudaMemcpy(d\_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU add<<<1,1>>>(d\_a, d\_b, d\_c);

// Copy result back to host

cudaError err = cudaMemcpy(&c, d\_c, size, cudaMemcpyDeviceToHost); if(err!=cudaSuccess) {

printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));

}

printf("result is %d\n",c);

// Cleanup cudaFree(d\_a); cudaFree(d\_b); cudaFree(d\_c); return 0;

}

# CUDA Program – 2

%%cu

#include <stdio.h> #define N 64

inline cudaError\_t checkCudaErr(cudaError\_t err, const char\* msg) { if (err != cudaSuccess) { fprintf(stderr, "CUDA Runtime error at %s: %s\n", 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 < N && col < N) { for (int i = 0; i < 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 < N; ++row ) for( int col = 0; col < N; ++col )

{

val = 0;

for ( int k = 0; k < 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 (&a, size); cudaMallocManaged (&b, size); cudaMallocManaged (&c\_cpu, size); cudaMallocManaged (&c\_gpu, size);

// Initialize memory; create 2D matrices for( int row = 0; row < N; ++row ) for( int col = 0; col

< 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 <<< number\_of\_blocks, threads\_per\_block >>> ( a, b, c\_gpu ); checkCudaErr(cudaDeviceSynchronize(), "Syncronization");

checkCudaErr(cudaGetLastError(), "GPU");

// 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 < N && !error; ++row ) for( int col = 0; col < N && !error; ++col ) if (c\_cpu[row \* N + col] != c\_gpu[row \* N + col])

{

printf("FOUND ERROR at c[%d][%d]\n", row, col); error = true; break;

}

if (!error) printf("Success!\n");

// Free all our allocated memory cudaFree(a); cudaFree(b); cudaFree( c\_cpu ); cudaFree( c\_gpu );

}

# CUDA Program – 3

%%cu #include<stdio.h> #include<cuda.h> int main()

{

cudaDeviceProp p; int device\_id; int major; int minor;

cudaGetDevice(&device\_id); cudaGetDeviceProperties(&p,device\_id); major=p.major; minor=p.minor;

printf("Name of GPU on your system is %s\n",p.name);

printf("\n Compute Capability of a current GPU on your system is %d.%d",majo r,minor);

return 0;

}

# CUDA Program – 4

%%cu #include<stdio.h> #include<cuda.h>

#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<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 \*\*)&d,row1\*col1\*sizeof(int)); cudaMalloc((void

\*\*)&e,row2\*col2\*sizeof(int)); cudaMalloc((void \*\*)&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<<<grid,col1>>>(d,e,f);

cudaMemcpy(c,f,row1\*col2\*sizeof(int),cudaMemcpyDeviceToHost); printf("\n Product of two matrices:\n "); for(i=0;i<row1;i++)

{

for(j=0;j<col2;j++)

{

printf("%d\t",c[i][j]);

}

printf("\n");

}

cudaFree(d); cudaFree(e); cudaFree(f); return 0;

}

# CUDA Program – 5

%%cu #include<stdio.h> #include<cuda.h> 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 \*\*)&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<<<1,5>>>(c); cudaMemcpy(a,c,5\*sizeof(int),cudaMemcpyDeviceToHost);

printf("Elements of your array after addition with constant array {1,2,3,4,5} :\n")

;

for(i=0;i<5;i++)

{

printf("%d\t",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("float32") / 255 x\_test = x\_test.astype("float32") / 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("x\_train shape:", x\_train.shape) print(x\_train.shape[0], "train samples") print(x\_test.shape[0], "test samples")

# 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="relu"), layers.MaxPooling2D(pool\_size=(2, 2)), layers.Conv2D(64, kernel\_size=(3, 3), activation="relu"),

layers.MaxPooling2D(pool\_size=(2, 2)), layers.Flatten(), layers.Dropout(0.5), layers.Dense(num\_classes, activation="softmax"),

]

)

model.summary()

import datetime print("Training the Model ")

start\_time=datetime.datetime.now() print("Training started at: {}".format(start\_time)) batch\_size = 128

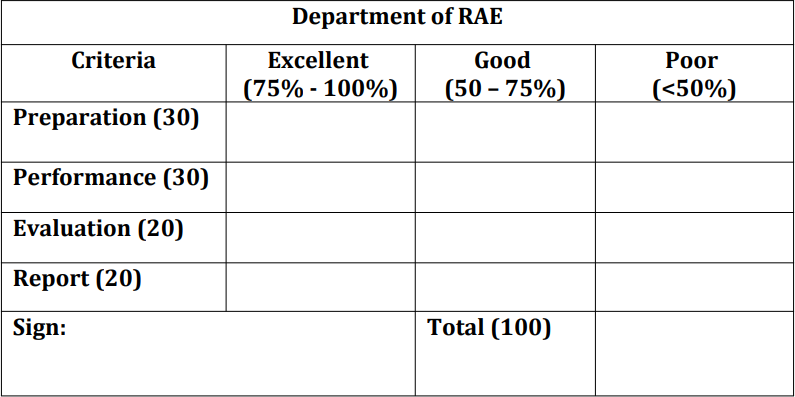
epochs = 15

model.compile(loss="categorical\_crossentropy", optimizer="adam", metrics=["accuracy"])

model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_split=0.1) end\_time= datetime.datetime.now() print("Training ended at: {}".format(end\_time)) duration

= end\_time - start\_time print("Training Duration: {}".format(duration))

score = model.evaluate(x\_test, y\_test, verbose=0) print("Test loss:", score[0])



# Result:

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