Assignment: 04

Matrix Multiplication

Code:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <cstdio>
#include <ctime>
#include <stdio.h>
#include <stdlib.h> #include <time.h>

// Kernel code
__global__ void calc_prod_cuda(int* A, int* B, int* C, int rows_a, int cols_a, int rows_b, int cols_b) {
  // get row, column from block and therd index   int g = blockIdx.x *
blockDim.x + threadIdx.x;
  int col = g / rows_a, row = g % rows_a;

  // calcuate prod for a cell   C[row * rows_b + col]
= 0;   for (int i = 0; i < cols_b; i++) {
    C[row * cols_b + col] += A[row * cols_a + i]*B[i * cols_b + col];
  }
}

// serial prouduct method
void calc_prod_serial(int * A, int* B, int* C, int rows_a, int cols_a, int rows_b, int cols_b) {
  // traverse rows
  for (int i=0; i < rows_a; i++) {     // traverse
column     for (int j=0; j < cols_b; j++) {
      // calcuate prod for a cell       C[i * cols_b +j] = 0;
for (int k=0; k < cols_b; k++) {
        C[i * cols_b + j] += A[i * cols_a + k] * B[k * cols_b + j];
      }
    }
  }
}

void initialize_matrix(
  int *host_a, int *host_b, int *host_prod, // Host matrices
  int rows_a, int cols_a, // dimenstin of A
  int rows_b, int cols_b // dimensions of B
) {
  printf("Initializing matrix..\n");

  //initialize A, B
  for (int i = 0; i < rows_a * cols_a; i++) {
    host_a[i] = i;
```

```
  }
  for (int i = 0; i < rows_b * cols_b; i++) {     host_b[i] = i+i;
  }

  printf("Matrix initialized\n");   fflush(stdout); }

// function of print matrix
void display_matrix(int *matrix, int rows, int cols) {
  for (int i = 0; i < rows; i++) {     for (int j = 0; j < cols;
j++) {
      printf("%d ", matrix[i * cols + j]);
    }
    printf("\n");
  }
}

// gpu matrix multiplication function void calculate_cuda(
  int *host_a, int *host_b, int *host_prod, // Host matrices   int rows_a, int
cols_a, // dimenstin of A   int rows_b, int cols_b, // dimensions of B
  int rows_prod, int cols_prod, // dimensions of prod
  bool show_product
) {

 // initialize matrix on device   int *device_a, *device_b,

*device_prod;   printf("\nCalculating PARALLEL..\n");

  // Allocate on device
  cudaMalloc((void**) &device_a, rows_a * cols_a * sizeof(int));   cudaMalloc((void**) &device_b,
rows_b * rows_b * sizeof(int));   cudaMalloc((void**) &device_prod, rows_prod * cols_prod *
sizeof(int));

  // Copy host to device   cudaMemcpy(
    device_a, host_a,     rows_a * rows_b * sizeof(int),
    cudaMemcpyHostToDevice
  );
  cudaMemcpy(
    device_b, host_b,
    rows_b * cols_b * sizeof(int),     cudaMemcpyHostToDevice
  );

  // Define grid and block dimensions   dim3 blockDim(cols_b);
  dim3 gridDim(rows_a);

  clock_t start_time = clock();

  // multiply
  calc_prod_cuda <<<gridDim, blockDim>>> (     device_a, device_b,
device_prod,
    rows_a, cols_a,
    rows_b, cols_b
  );
```

```
  // Copy the result back to the host
  cudaMemcpy(     host_prod, device_prod,
    rows_prod * cols_prod * sizeof(int),
    cudaMemcpyDeviceToHost
  );

  if (show_product) {     printf("\nProduct is:\n");
display_matrix(host_prod, rows_prod, cols_prod);
  }

  printf(
    "\nProduct calculated in %f seconds\n",
    (double)(clock() - start_time) / CLOCKS_PER_SEC
  );
  fflush(stdout);

  cudaFree(device_a);   cudaFree(device_b);
  cudaFree(device_prod);
}

// serial matrix multiplication function void calculate_serial(
  int *host_a, int *host_b, int *host_prod, // Host matrices   int rows_a, int
cols_a, // dimenstin of A   int rows_b, int cols_b, // dimensions of B
  int rows_prod, int cols_prod, // dimensions of prod
  bool show_product
) {
  clock_t start_time = clock();   printf("\nCalculating Serial..\n");

  calc_prod_serial(     host_a, host_b, host_prod,
rows_a, rows_b,
    rows_b, cols_b
  );
  if (show_product) {     printf("\nProduct is:\n");
    display_matrix(host_prod, rows_prod, cols_prod);
  }
  printf(
    "\nProduct calculated in %f seconds\n",
    (double)(clock() - start_time) / CLOCKS_PER_SEC
  );
  fflush(stdout); }

void free_matrix(int *host_a, int *host_b, int *host_prod) {
  // free memory   free(host_a);   free(host_b);
free(host_prod);
}

int main() {   int i=1;   while (true) {
    if (i==1) {
      int rows_a, cols_a, rows_b, cols_b, see_prod;

      printf("\nEnter dimensions of Matrix: ");       scanf("%d", &rows_a);
```

```
      cols_a =  cols_b = rows_b = rows_a;

      printf("\nDo you want to see prouct? ");       scanf("%d", &see_prod);
      printf("\n");

      int *A, *B, *prod;

      // matrix size       int rows_prod = rows_a;
int cols_prod = cols_b;

      // allocate on host
      A = (int*) malloc (rows_a * cols_a * sizeof(int));       B = (int*) malloc (rows_b *
cols_b * sizeof(int));       prod = (int*) malloc (rows_prod * cols_prod * sizeof(int));

      initialize_matrix(        A, B, prod,
rows_a, cols_a,
        rows_b, cols_b
      );
      calculate_cuda(        A, B, prod,        rows_a,
cols_a,        rows_b, cols_b,        rows_prod,
cols_prod,        see_prod
      );
      calculate_serial(        A, B, prod,
rows_a, cols_a,        rows_b, cols_b,
rows_prod, cols_prod,
        see_prod
      );

      free_matrix(A, B, prod);
    } else {       break;
    }
    printf("Enter 1 to calculate again? ");     scanf("%d", &i);
  }
}
```

Output:

```
Enter dimensions of Matrix: 4

Do you want to see prouct? 1

Initializing matrix..
Matrix initialized

Calculating PARALLEL..

Product is:
112 124 136 148
304 348 392 436
496 572 648 724
688 796 904 1012

Product calculated in 0.000116 seconds

Calculating Serial..

Product is:
112 124 136 148
304 348 392 436
496 572 648 724
688 796 904 1012

Product calculated in 0.000010 seconds
Enter 1 to calculate again? 1

Enter dimensions of Matrix: 2000

Do you want to see prouct? 0

Initializing matrix..
Matrix initialized

Calculating PARALLEL..

Product calculated in 0.003006 seconds

Calculating Serial..

Product calculated in 33.370725 seconds
Enter 1 to calculate again? 0
```

Vector Addition

Code:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <ctime>
#include <iostream> #include <time.h> using
namespace std;

__global__ void add(int* A, int* B, int* C, int size) {     int tid = blockIdx.x *
blockDim.x + threadIdx.x;

    if (tid < size) {         C[tid] = A[tid] + B[tid];
    }
}

void add_serial(int *A, int *B, int*C, int size) {   for (int i=0; i<size; i++)
{
    C[i] = A[i] + B[i];
  }
}


void initialize(int* vector, int size) {     for (int i = 0; i < size;
i++) {
        vector[i] = rand() % 10;
    }
}

void print(int* vector, int size) {     for (int i = 0; i < size;
i++) {
        cout << vector[i] << " ";
    }
    cout << endl;
}

int main() {   int i = 1;   while (i == 1) {
int N = 4;     int* A, * B, * C;

    int vectorSize;
    cout << "\nEnter size of Vector: ";     cin >> vectorSize;
    size_t vectorBytes = vectorSize * sizeof(int);
```

```
    A = new int[vectorSize];     B = new
int[vectorSize];     C = new int[vectorSize];

    bool shoulprint;     cout << "\nDisplay Vectors? ";
cin >> shoulprint;

    initialize(A, vectorSize);     initialize(B,

vectorSize);     if (shoulprint) {

      cout << "\nVector A: ";       print(A, N);
cout << "Vector B: ";       print(B, N);
    }

    cout << "\nCalculating Parallel..\n"     clock_t start_time =
clock();

    int* X, * Y, * Z;     cudaMalloc(&X, vectorBytes);
cudaMalloc(&Y, vectorBytes);     cudaMalloc(&Z,
vectorBytes);

    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);     cudaMemcpy(Y, B, vectorBytes,
cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;     add<<<blocksPerGrid,

threadsPerBlock>>>(X, Y, Z, N);     cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

    if (shoulprint) {       cout << "Addition: ";
print(C, N);
    }
    cout << "Time taken: " << (double) (clock() - start_time) / CLOCKS_PER_SEC << "\n\n";     cout << "Calculating

Serial..\n";

    start_time = clock()     add_serial(A, B, C, vectorSize);

    if (shoulprint) {       cout << "Addition: ";
      print(C, vectorSize);
    }
    cout << "Time taken: " << (double) (clock() - start_time) / CLOCKS_PER_SEC << "\n\n";


    delete[] A;     delete[] B;     delete[] C;

    cudaFree(X);     cudaFree(Y);     cudaFree(Z);

    cout << "Enter 1 to go again: ";     cin >> i;
  }
  return 0;
}
```

Output:

```
Enter size of Vector: 5

Display Vectors? 1

Vector A: 3 6 7 5
Vector B: 5 6 2 9

Calculating Parallel..
Addition: 8 12 9 14
Time taken: 0.127618

Calculating Serial..
Addition: 8 12 9 14 4
Time taken: 2e-06

Enter 1 to go again: 1

Enter size of Vector: 10000000

Display Vectors? 0

Calculating Parallel..
Time taken: 0.034184

Calculating Serial..
Time taken: 0.023275

Enter 1 to go again: 1

Enter size of Vector: 1000000000

Display Vectors? 0

Calculating Parallel..
Time taken: 0.019662

Calculating Serial..
Time taken: 2.56112

Enter 1 to go again: 0
```

Assignment-5

[1]:
```python
import tensorflow as tf

# Display the version
print(tf.__version__)

# other imports
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout
from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model
```

```
2.15.0
```

[2]:
```python
# Load in the data
cifar10 = tf.keras.datasets.cifar10

# Distribute it to train and test set
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(50000, 32, 32, 3) (50000, 1) (10000, 32, 32, 3) (10000, 1)
```

[3]:
```python
# Reduce pixel values
x_train, x_test = x_train / 255.0, x_test / 255.0

# flatten the label values
y_train, y_test = y_train.flatten(), y_test.flatten()
```

[4]:
```python
# number of classes
K = len(set(y_train))

# calculate total number of classes
# for output layer
print("number of classes:", K)

# Build the model using the functional API
```

```
# input layer
i = Input(shape=x_train[0].shape)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(i)
x = BatchNormalization()(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = BatchNormalization()(x)
x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)
x = Dropout(0.2)(x)

# Hidden layer
x = Dense(1024, activation='relu')(x)
x = Dropout(0.2)(x)

# last hidden layer i.e.. output layer
x = Dense(K, activation='softmax')(x)

model = Model(i, x)

# model description
model.summary()
```

```
number of classes: 10
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 32)        896

 batch_normalization (Batch  (None, 32, 32, 32)        128
 Normalization)
```

conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147584
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 1024)	2098176
dropout_1 (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 10)	10250

===
Total params: 2397226 (9.14 MB)
Trainable params: 2396330 (9.14 MB)
Non-trainable params: 896 (3.50 KB)

[5]:
```
# Compile
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

[6]:
```
# Fit
r = model.fit(
x_train, y_train, validation_data=(x_test, y_test), epochs=10)
```

```
Epoch 1/10
1563/1563 [==============================] - 20s 8ms/step - loss: 1.3153 -
accuracy: 0.5517 - val_loss: 1.7805 - val_accuracy: 0.4983
Epoch 2/10
1563/1563 [==============================] - 11s 7ms/step - loss: 0.8354 -
accuracy: 0.7099 - val_loss: 1.0786 - val_accuracy: 0.6472
Epoch 3/10
1563/1563 [==============================] - 11s 7ms/step - loss: 0.6803 -
accuracy: 0.7652 - val_loss: 0.7639 - val_accuracy: 0.7428
Epoch 4/10
1563/1563 [==============================] - 12s 8ms/step - loss: 0.5806 -
accuracy: 0.8004 - val_loss: 0.7259 - val_accuracy: 0.7563
Epoch 5/10
1563/1563 [==============================] - 12s 8ms/step - loss: 0.4966 -
accuracy: 0.8290 - val_loss: 0.7507 - val_accuracy: 0.7548
Epoch 6/10
1563/1563 [==============================] - 11s 7ms/step - loss: 0.4200 -
accuracy: 0.8546 - val_loss: 0.6544 - val_accuracy: 0.7887
Epoch 7/10
1563/1563 [==============================] - 18s 11ms/step - loss: 0.3475 -
accuracy: 0.8789 - val_loss: 0.6331 - val_accuracy: 0.8042
Epoch 8/10
1563/1563 [==============================] - 12s 7ms/step - loss: 0.2996 -
accuracy: 0.8971 - val_loss: 0.6709 - val_accuracy: 0.8032
Epoch 9/10
1563/1563 [==============================] - 12s 8ms/step - loss: 0.2601 -
accuracy: 0.9097 - val_loss: 0.6870 - val_accuracy: 0.8097
Epoch 10/10
1563/1563 [==============================] - 11s 7ms/step - loss: 0.2205 -
accuracy: 0.9233 - val_loss: 0.6497 - val_accuracy: 0.8206
```

[7]:
```
# Plot accuracy per iteration
plt.plot(r.history['accuracy'], label='acc', color='red')
plt.plot(r.history['val_accuracy'], label='val_acc', color='green')
plt.legend()
```

[7]: <matplotlib.legend.Legend at 0x7e21f398eb30>