HPC_assignment_17

April 16, 2024

1 HPC assignment 17

[ ]:
```
!nvidia-smi
```

```
Tue Apr 16 07:06:05 2024
+-----------------------------------------------------------------------------------
--------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version:
12.2     |
|-----------------------------------------+----------------------+--------------
--------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+==============
========|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   40C    P8               9W /  70W |      0MiB / 15360MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------
--------+

+-----------------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                           GPU
Memory |
|        ID   ID
Usage      |
|===================================================================================
========|
```

```
|  No running processes found
|
+-----------------------------------------------------------------------------
---------+
```

1.1 1. To print hello message on the screen using kernal function

[ ]:
```
%%writefile hello_1_1.cu

#include <stdio.h>

__global__ void cuda_hello_1_1() {
    printf("Hello World from GPU with grid dimension (1, 1) and block dimension␣
↪(1, 1)!\n");
}

int main() {
    cuda_hello_1_1<<<1,1>>>();
    cudaDeviceSynchronize(); // Make sure all GPU work is done before exiting
    return 0;
}
```

```
Writing hello_1_1.cu
```

[ ]:
```
!nvcc -o hello_1_1 hello_1_1.cu
```

[ ]:
```
!./hello_1_1
```

```
Hello World from GPU with grid dimension (1, 1) and block dimension (1, 1)!
```

1.2 2. To add two vectors of size 100 and 20000 and analyze the performance comparison between cpu and gpu processing

1.2.1 GPU

[ ]:
```
!pip install pycuda
```

```
Collecting pycuda
  Downloading pycuda-2024.1.tar.gz (1.7 MB)
                           1.7/1.7 MB
12.0 MB/s eta 0:00:00
  Installing build dependencies … done
  Getting requirements to build wheel … done
  Preparing metadata (pyproject.toml) … done
Collecting pytools>=2011.2 (from pycuda)
  Downloading pytools-2024.1.1-py2.py3-none-any.whl (85 kB)
                           85.1/85.1 kB
12.0 MB/s eta 0:00:00
Requirement already satisfied: appdirs>=1.4.0 in
```

```
/usr/local/lib/python3.10/dist-packages (from pycuda) (1.4.4)
Collecting mako (from pycuda)
  Downloading Mako-1.3.3-py3-none-any.whl (78 kB)
                           78.8/78.8 kB
10.9 MB/s eta 0:00:00
Requirement already satisfied: platformdirs>=2.2.0 in
/usr/local/lib/python3.10/dist-packages (from pytools>=2011.2->pycuda) (4.2.0)
Requirement already satisfied: typing-extensions>=4.0 in
/usr/local/lib/python3.10/dist-packages (from pytools>=2011.2->pycuda) (4.11.0)
Requirement already satisfied: MarkupSafe>=0.9.2 in
/usr/local/lib/python3.10/dist-packages (from mako->pycuda) (2.1.5)
Building wheels for collected packages: pycuda
  Building wheel for pycuda (pyproject.toml) … done
  Created wheel for pycuda: filename=pycuda-2024.1-cp310-cp310-linux_x86_64.whl
size=661204
sha256=51efb7c5582dd86e48b9404a05e0a366352406f4840bf4dc162fe9a89aa2ad1c
  Stored in directory: /root/.cache/pip/wheels/12/34/d2/9a349255a4eca3a486d82c79
d21e138ce2ccd90f414d9d72b8
Successfully built pycuda
Installing collected packages: pytools, mako, pycuda
Successfully installed mako-1.3.3 pycuda-2024.1 pytools-2024.1.1
```

[ ]:
```
import numpy as np
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import time
```

[ ]:
```
# CUDA kernel function to add two vectors
cuda_kernel_code = """
__global__ void vector_add(float *a, float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
"""
```

[ ]:
```
# Compile the CUDA kernel code
cuda_module = SourceModule(cuda_kernel_code)

# Get a reference to the CUDA kernel function
vector_add_cuda = cuda_module.get_function("vector_add")
```

[ ]:
```
def vector_add_gpu(a, b):
    n = a.size

    # Create device arrays
```

```
    a_gpu = cuda.mem_alloc(a.nbytes)
    b_gpu = cuda.mem_alloc(b.nbytes)
    c_gpu = cuda.mem_alloc(b.nbytes)

    # Copy data to device
    cuda.memcpy_htod(a_gpu, a)
    cuda.memcpy_htod(b_gpu, b)

    # Define block and grid dimensions
    block_dim = (256, 1, 1)
    grid_dim = ((n + block_dim[0] - 1) // block_dim[0], 1)

    start_time = time.time()

    # Launch the CUDA kernel
    vector_add_cuda(a_gpu, b_gpu, c_gpu, np.int32(n), block=block_dim,␣
  ↪grid=grid_dim)

    # Synchronize threads to ensure all output is calculated
    cuda.Context.synchronize()

    end_time = time.time()

    # Copy result back to host
    c = np.empty_like(a)
    cuda.memcpy_dtoh(c, c_gpu)

    return c, end_time - start_time
```

[ ]:
```
vector_size_1 = 100
vector_size_2 = 20000
a = np.random.randn(vector_size_2).astype(np.float32)
b = np.random.randn(vector_size_2).astype(np.float32)

# Perform vector addition on GPU
result_gpu1, gpu_time1 = vector_add_gpu(a[:vector_size_1], b[:vector_size_1])
result_gpu2, gpu_time2 = vector_add_gpu(a[:vector_size_2], b[:vector_size_2])
```

[ ]:
```
print("Vector addition of size", vector_size_1, "on GPU took", gpu_time1,␣
  ↪"seconds.")

print("Vector addition of size", vector_size_2, "on GPU took", gpu_time2,␣
  ↪"seconds.")
```

```
Vector addition of size 100 on GPU took 0.0007643699645996094 seconds.
Vector addition of size 20000 on GPU took 6.818771362304688e-05 seconds.
```

1.2.2 CPU

[ ]:
```python
import numpy as np
import time
```

[ ]:
```python
def vector_add_cpu(a, b):
    start_time = time.time()
    result = a + b
    end_time = time.time()
    return result, end_time - start_time
```

[ ]:
```python
vector_size_1 = 100
vector_size_2 = 20000
a = np.random.randn(vector_size_2).astype(np.float32)
b = np.random.randn(vector_size_2).astype(np.float32)

# Perform vector addition on CPU
result_cpu1, cpu_time1 = vector_add_cpu(a[:vector_size_1], b[:vector_size_1])
result_cpu2, cpu_time2 = vector_add_cpu(a[:vector_size_2], b[:vector_size_2])
```

[ ]:
```python
print("Vector addition of size", vector_size_1, "on CPU took", cpu_time1,␣
 ↪"seconds.")
print("Vector addition of size", vector_size_2, "on CPU took", cpu_time2,␣
 ↪"seconds.")
```

```
Vector addition of size 100 on CPU took 2.3365020751953125e-05 seconds.
Vector addition of size 20000 on CPU took 1.9311904907226562e-05 seconds.
```

- Vector addition of size 100 on CPU took 2.384185791015625e-05 seconds.
- Vector addition of size 20000 on CPU took 1.9788742065429688e-05 seconds.

- Vector addition of size 100 on GPU took 0.0007691383361816406 seconds.
- Vector addition of size 20000 on GPU took 7.128715515136719e-05 seconds.

1.3 3. To multply two matrix of size 20 X 20 and 1024 X 1024 analyze the performance comparison between cpu and gpu processing

1.3.1 GPU

[ ]:
```python
def matrix_multiply_gpu(a, b):
    # Define CUDA kernel code for matrix multiplication
    cuda_code = """
    __global__ void matrix_multiply(float *a, float *b, float *c, int n) {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if (row < n && col < n) {
            float sum = 0.0;
            for (int i = 0; i < n; ++i) {
                sum += a[row * n + i] * b[i * n + col];
```

```
            }
            c[row * n + col] = sum;
        }
    }
    """

    # Compile CUDA kernel code
    mod = SourceModule(cuda_code)

    # Get kernel function
    matrix_multiply_cuda = mod.get_function("matrix_multiply")

    # Allocate memory on device
    a_gpu = cuda.mem_alloc(a.nbytes)
    b_gpu = cuda.mem_alloc(b.nbytes)
    c_gpu = cuda.mem_alloc(a.nbytes)

    # Copy input matrices to device
    cuda.memcpy_htod(a_gpu, a)
    cuda.memcpy_htod(b_gpu, b)

    # Define grid and block dimensions
    block_size = (16, 16, 1)
    grid_size = ((a.shape[1] + block_size[0] - 1) // block_size[0], (a.shape[0]␣
↪+ block_size[1] - 1) // block_size[1], 1)

    # Call CUDA kernel
    matrix_multiply_cuda(a_gpu, b_gpu, c_gpu, np.int32(a.shape[0]),␣
↪block=block_size, grid=grid_size)

    # Copy result back to host
    c = np.empty_like(a)
    cuda.memcpy_dtoh(c, c_gpu)

    return c
```

[ ]:
```
# Function to generate random matrices
def generate_random_matrix(rows, cols):
    return np.random.rand(rows, cols).astype(np.float32)
```

[ ]:
```
# Function to measure time taken for matrix multiplication
def measure_time(matrix_size, func, *args):
    start_time = time.time()
    result = func(*args)
    end_time = time.time()
    return result, end_time - start_time
```

```
[ ]: # Sizes of matrices to be multiplied
matrix_sizes = [(20, 20), (1024, 1024)]
```

```
[ ]: for size in matrix_sizes:
    print(f"\nMatrix size: {size}")
    a = generate_random_matrix(*size)
    b = generate_random_matrix(*size)

    # GPU matrix multiplication
    gpu_result, gpu_time = measure_time(size, matrix_multiply_gpu, a, b)
    print(f"GPU time: {gpu_time:.6f} seconds")
```

```
Matrix size: (20, 20)
GPU time: 0.428407 seconds

Matrix size: (1024, 1024)
GPU time: 0.018636 seconds
```

1.3.2 CPU

```
[ ]: # CPU matrix multiplication
def matrix_multiply_cpu(a, b):
    result = np.zeros((a.shape[0], b.shape[1]), dtype=np.float32)
    for i in range(a.shape[0]):
        for j in range(b.shape[1]):
            for k in range(a.shape[1]):
                result[i, j] += a[i, k] * b[k, j]
    return result
```

```
[ ]: # Function to generate random matrices
def generate_random_matrix(rows, cols):
    return np.random.rand(rows, cols).astype(np.float32)
```

```
[ ]: # Function to measure time taken for matrix multiplication
def measure_time(matrix_size, func, *args):
    start_time = time.time()
    result = func(*args)
    end_time = time.time()
    return result, end_time - start_time
```

```
[ ]: # Sizes of matrices to be multiplied
matrix_sizes = [(20, 20), (1024, 1024)]
```

```
[ ]: for size in matrix_sizes:
    print(f"\nMatrix size: {size}")
    a = generate_random_matrix(*size)
    b = generate_random_matrix(*size)
```

```
    # CPU matrix multiplication
    cpu_result, cpu_time = measure_time(size, matrix_multiply_cpu, a, b)
    print(f"CPU time: {cpu_time:.6f} seconds")
```

```
Matrix size: (20, 20)
CPU time: 0.004824 seconds

Matrix size: (1024, 1024)
CPU time: 704.230331 seconds
```

- CPU Time for 1024: 0.12308359146118164 seconds
- CPU Time for 20: 0.0019140243530273438 seconds
- Matrix size: (20, 20)
- GPU time: 0.703994 seconds
- Matrix size: (1024, 1024)
- GPU time: 0.014648 seconds

1.4 4. To obtain CUDA device information and print the output

[ ]:
```
import pycuda.driver as cuda

# Initialize PyCUDA
cuda.init()

# Get the number of CUDA devices
num_devices = cuda.Device.count()

print("Number of CUDA devices:", num_devices)

# Iterate over each CUDA device and print its properties
for i in range(num_devices):
    device = cuda.Device(i)
    print("\nCUDA Device:", i)
    print("  Name:", device.name())
    print("  Compute Capability:", device.compute_capability())
    print("  Total Memory:", device.total_memory() / (1024 ** 3), "GB")
    print("  Max Threads per Block:", device.max_threads_per_block)
    print("  Multiprocessor Count:", device.multiprocessor_count)
    print("  Clock Rate:", device.clock_rate / 1e6, "GHz")
```

```
Number of CUDA devices: 1

CUDA Device: 0
  Name: Tesla T4
  Compute Capability: (7, 5)
```

```
Total Memory: 14.74810791015625 GB
Max Threads per Block: 1024
Multiprocessor Count: 40
Clock Rate: 1.59 GHz
```