
# IT DATA SECURITY LAB FILE

**Name- Dhairya Jain**
**Sap ID- 500105432**
**Batch- CSF-B4**

**EXPERIMENT-11**

## Part A- GPU Parallel Programming

### Example Using CuPy

- Install CuPy

```
D:\data sec>pip install cupy-cuda11x
Requirement already satisfied: cupy-cuda11x in c:\python312\lib\site-package
s (13.3.0)
Requirement already satisfied: numpy<2.3,>=1.22 in c:\python312\lib\site-pac
kages (from cupy-cuda11x) (2.0.2)
Requirement already satisfied: fastrlock>=0.5 in c:\python312\lib\site-packa
ges (from cupy-cuda11x) (0.8.2)

[notice] A new release of pip is available: 24.0 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

- GPU Parallel Programming Example Using CuPy

Code-

exp11_1.py | exp11_2.py | hello.py

exp11_1.py > ...

```python
import cupy as cp
import numpy as np
import time
# Define the size of the arrays
n = 10000000
# Create two large arrays on the GPU
a_gpu = cp.random.rand(n)
b_gpu = cp.random.rand(n)
# Perform the addition on the GPU
start_gpu = time.time()
c_gpu = a_gpu + b_gpu
end_gpu = time.time()
# Transfer the result back to the CPU (if needed)
c_cpu = cp.asnumpy(c_gpu)
print("GPU addition took:", end_gpu - start_gpu, "seconds")
# For comparison, perform the same operation on the CPU using NumPy
a_cpu = np.random.rand(n)
b_cpu = np.random.rand(n)
start_cpu = time.time()
c_cpu = a_cpu + b_cpu
end_cpu = time.time()
print("CPU addition took:", end_cpu - start_cpu, "seconds")
```

Output-

```
PS D:\data sec> python -u "d:\data sec\exp11_1.py"
GPU addition took: 0.006550788879394531 seconds
CPU addition took: 0.027991056442260742 seconds
PS D:\data sec>
```

- Explanation:
  - Creating Arrays: Two large random arrays are created directly on the GPU using CuPy.
  - Parallel Addition: The addition operation is performed on the GPU, leveraging its parallel processing capabilities.
  - Performance Comparison: The script also performs the same operation on the CPU using NumPy to compare performance.

**Example Using Numba**

- Install Numba and CUDA Toolkit

```
D:\data sec>pip install numba
Requirement already satisfied: numba in c:\python312\lib\site-packages (0.60
.0)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in c:\python312\li
b\site-packages (from numba) (0.43.0)
Requirement already satisfied: numpy<2.1,>=1.22 in c:\python312\lib\site-pac
kages (from numba) (2.0.0)

[notice] A new release of pip is available: 24.0 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

- GPU Parallel Programming Example Using Numba
  Code-

exp11_1.py | exp11_2.py | hello.py

exp11_2.py > vector_add

```python
import numpy as np
from numba import cuda
import time
# Define the size of the arrays
n = 10000000
# Initialize arrays on the CPU
a_cpu = np.random.rand(n)
b_cpu = np.random.rand(n)
c_cpu = np.zeros(n)
# Define a GPU kernel for vector addition
@cuda.jit
def vector_add(a, b, c):
    idx = cuda.grid(1)
    if idx < a.size:
        c[idx] = a[idx] + b[idx]
# Allocate memory on the GPU
a_gpu = cuda.to_device(a_cpu)
b_gpu = cuda.to_device(b_cpu)
c_gpu = cuda.device_array_like(a_cpu)
# Define the number of threads and blocks
threads_per_block = 1024
blocks_per_grid = (a_cpu.size + (threads_per_block - 1)) // threads_per_block
# Perform the addition on the GPU
start_gpu = time.time()
vector_add[blocks_per_grid, threads_per_block](a_gpu, b_gpu, c_gpu)
cuda.synchronize()
end_gpu = time.time()
# Transfer the result back to the CPU
c_cpu = c_gpu.copy_to_host()
print("GPU addition took:", end_gpu - start_gpu, "seconds")
# For comparison, perform the same operation on the CPU
start_cpu = time.time()
c_cpu = a_cpu + b_cpu
end_cpu = time.time()
print("CPU addition took:", end_cpu - start_cpu, "seconds")
```

Output-

```
PS D:\data sec> python -u "d:\data sec\exp11_2.py"
GPU addition took: 0.19643712043762207 seconds
CPU addition took: 0.027849912643432617 seconds
PS D:\data sec>
```

- Explanation:
  - Defining a GPU Kernel: The vector_add function is decorated with @cuda.jit, indicating it should be compiled to run on the GPU. The cuda.grid(1) function provides the unique index of each thread in the 1D grid.
  - Memory Management: Arrays are transferred from the CPU to the GPU and vice versa using cuda.to_device() and cuda.device_array_like().
  - Kernel Execution: The kernel is launched with a specific number of blocks and threads, allowing it to run in parallel on the GPU.
  - Synchronization: cuda.synchronize() ensures that the GPU has completed all its work before measuring the execution time.