DYP

Assignment No 4.

Title : CUDA Program.

Objective : Is to demonstrate how to use CUDA C to perform vector addition and matrix multiplication on the GPU.

Problem Statement : Write a CUDA program for
1. Addition of two large vectors.
2. Matrix Multiplication using CUDA C.

Theory :

Programming Structure of GPU and CPU.
CUDA kernel:
The function which are executed on GPU are called as kernels. Kernels are full program or function invoke by the CPU and executed on GPU. A kernal is executed N number of times in parallel on GPU by using N number of threads.
Invocation: kernel_name<<< grid, block >>>(argument list);
Kernel is defined as:

```
__global__ void kernel_name (arguments)
{
    ....
}
```

Parallel Vector Addition:
Procedure:
1) Write a program using text editor, name the source code with .cu extension.

2) Compile the program using nvcc compiler.
3) Execute the program.
4) Verify the result.

- Adding of Two Large Vectors:

1. Vector addition is a simple yet computationally intensive operation that involves adding the corresponding elements of two vectors. In CUDA C, vector addition can be implemented by dividing the vectors into blocks and threads and performing the addition in parallel on the GPU.

The following steps are involved in implementing vector addition using CUDA C : Allocate memory on the device (GPU) using cudaMalloc()
1) Copy the input vectors from the host (CPU) to device using cudaMemcpy()
2) Launch the kernel on the GPU using the <<<>>>
3) Wait for the kernel to finish using cudaDeviceSynchronize().
4) Copy the result back from device to host using cudaMemcpy().
5) Free the memory on the device using cudaFree().

- Matrix Multiplication:

Matrix Multiplication is a fundamental operation in linear algebra that involves multiplying two matrices to produce a third matrix. It is a computationally intensive operation that can benefit from parallelization on GPUs.

The following steps are involved in implementing matrix multiplication using CUDA C:

1) Allocate memory on the device (GPU) using cudaMalloc().
2) Copy the input matrices from host (CPU) to device using cudaMemCpy()
3) Launch the kernel on the GPU using <<<>>> syntax.
4) Wait for the kernel to finish using cudaDeviceSynchronize()
5) Free the memory on the device using cudaFree().

Conclusion: We have successfully implemented a recurrent neural network to create a classifier.