Experiment No.08
Implement Two Vector addition using OpenCL/CUDA/ Parallel Matlab
Date of Performance:28/03/2024
Date of Submission:18/04/2024

Aim: The aim of this task is to implement two vector addition using OpenCL/CUDA/Parallel Matlab.

Objective: The objective of this task is to implement two vector addition using parallel programming frameworks such as OpenCL, CUDA, or Parallel Matlab. Vector addition is a simple operation that can be performed in parallel, making it a good candidate for parallel computing using GPUs. By implementing two vector addition using these frameworks, we can demonstrate the benefits of parallel programming and compare the performance of different frameworks.

Theory:

Vector addition is a basic mathematical operation that adds two vectors element-wise to produce a third vector. It is a simple operation that can be easily parallelized across multiple processing elements, making it an ideal candidate for parallel programming**.**

OpenCL (Open Computing Language) and CUDA (Compute Unified Device Architecture) are two popular parallel programming frameworks used to accelerate compute-intensive applications on GPUs. These frameworks allow the programmer to write code that can be executed in parallel across multiple threads or processing elements.

In OpenCL, the programmer writes a kernel function that is executed on the GPU, and the input data is transferred from the host (CPU) to the device (GPU) memory. The kernel function is executed in parallel across multiple work-items, which are mapped to processing elements on the GPU. The output data is then transferred back to the host memory.

Similarly, in CUDA, the programmer writes a kernel function that is executed on the GPU, and the input data is transferred to the device memory. The kernel function is executed in parallel across multiple threads, which are mapped to processing elements on the GPU. The output data is then transferred back to the host memory.

Parallel Matlab is another framework that supports parallel programming using multiple cores or GPUs. The programmer can use the parfor loop construct to parallelize the computation of the vector addition operation. The parfor loop automatically distributes the computation across multiple cores or GPUs, and the output data is combined at the end of the loop.

Overall, implementing two vector addition using OpenCL, CUDA, or Parallel Matlab can significantly improve the performance of the computation, especially for large vectors. Parallel programming frameworks allow the programmer to take advantage of the parallel processing capabilities of GPUs to accelerate compute-intensive applications.

Code:

```
#include <stdio.h>


// Kernel function to add two vectors
__global__ void vectorAdd(int *a, int *b, int *c, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < n) {
        c[tid] = a[tid] + b[tid];
    }
}


int main() {
    int n = 1000; // Size of the vectors
```

```
int *a, *b, *c; // Host vectors
int *d_a, *d_b, *d_c; // Device vectors

// Allocate memory for host vectors
a = (int*)malloc(n * sizeof(int));
b = (int*)malloc(n * sizeof(int));
c = (int*)malloc(n * sizeof(int));

// Initialize host vectors
for (int i = 0; i < n; i++) {
    a[i] = i;
    b[i] = i * 2;
}

// Allocate memory for device vectors
cudaMalloc(&d_a, n * sizeof(int));
cudaMalloc(&d_b, n * sizeof(int));
cudaMalloc(&d_c, n * sizeof(int));

// Copy host vectors to device
cudaMemcpy(d_a, a, n * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, n * sizeof(int), cudaMemcpyHostToDevice);
```

```
    // Define grid and block dimensions
    dim3 blockSize(256);
    dim3 gridSize((n + blockSize.x - 1) / blockSize.x);

    // Launch kernel
    vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copy result back to host
    cudaMemcpy(c, d_c, n * sizeof(int), cudaMemcpyDeviceToHost);

    // Display result
    for (int i = 0; i < 10; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Free host memory
```

```
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Output:

```
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
```

Conclusion: In conclusion, implementing two vector addition using parallel programming frameworks such as OpenCL, CUDA, or Parallel Matlab provides a significant performance improvement over traditional sequential programming. These frameworks allow the programmer to take advantage of the parallel processing capabilities of GPUs to accelerate compute-intensive applications.