Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

Link : Click IT

Experiment 2

Aim: Implementation of parallel quicksort [Hyper quick sort] using CUDA.

Theory :

QuickSort is a popular sorting algorithm used in parallel computing, which follows a divide-and-conquer approach to sort an array. The basic idea is to partition the array into smaller sub-arrays and then sort those sub-arrays.

In parallel computing, the partitioning process can be performed in parallel using multiple threads or processors, making the sorting process faster. The parallelization of quicksort algorithms can be done in two ways: recursive parallelization and iterative parallelization.

In the recursive parallelization method, the partitioning process is performed by dividing the array into smaller sub-arrays recursively until the sub-arrays are small enough to be sorted by a single thread. Each recursive call creates a new thread to perform the partitioning process on the sub-array.

In the iterative parallelization method, the partitioning process is performed iteratively using a single thread that divides the array into smaller sub-arrays. These sub-arrays are pushed onto a stack and then popped off the stack and sorted iteratively until the entire array is sorted.

Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

In both methods, the partitioning process is the most important part of the quicksort algorithm. It involves selecting a pivot element, and then partitioning the array such that all the elements less than the pivot are on the left side of the pivot, and all the elements greater than the pivot are on the right side of the pivot. This process can be performed in parallel by dividing the array into smaller sub-arrays and performing the partitioning process on each sub-array.

After the partitioning process, the two resulting sub-arrays are sorted recursively or iteratively using quicksort algorithm. Finally, the sorted sub-arrays are merged to get the sorted array.

Parallel implementation of quicksort algorithm is very useful in situations where the size of the array is very large, and sequential sorting algorithm takes too much time.

Code :

```
%%cu
#include <stdio.h>
#include <stdlib.h>

__device__ int d_size;

__global__ void partition (int *arr, int *arr_l, int *arr_h, int n)
{
    int z = blockIdx.x*blockDim.x+threadIdx.x;
    d_size = 0;
    __syncthreads();
    if (z<n)
      {
        int h = arr_h[z];
        int l = arr_l[z];
        int x = arr[h];
        int i = (l - 1);
        int temp;
        for (int j = l; j <= h- 1; j++)
          {
            if (arr[j] <= x)
```

```
            {
              i++;
              temp = arr[i];
              arr[i] = arr[j];
              arr[j] = temp;
            }
          }
        temp = arr[i+1];
        arr[i+1] = arr[h];
        arr[h] = temp;
        int p = (i + 1);
        if (p-1 > l)
          {
            int ind = atomicAdd(&d_size, 1);
            arr_l[ind] = l;
            arr_h[ind] = p-1;
          }
        if ( p+1 < h )
          {
            int ind = atomicAdd(&d_size, 1);
            arr_l[ind] = p+1;
            arr_h[ind] = h;
          }
      }
}
void quickSortIterative (int arr[], int l, int h)
{
    int lstack[ h - l + 1 ], hstack[ h - l + 1];

    int top = -1, *d_d, *d_l, *d_h;

    lstack[ ++top ] = l;
    hstack[ top ] = h;

    cudaMalloc(&d_d, (h-l+1)*sizeof(int));
    cudaMemcpy(d_d, arr,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);

    cudaMalloc(&d_l, (h-l+1)*sizeof(int));
    cudaMemcpy(d_l, lstack,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);
```

Name : Manthan Gopal Dhole **TY IT**
Id : 201080020 **Parallel Computing**

```
    cudaMalloc(&d_h, (h-l+1)*sizeof(int));
    cudaMemcpy(d_h, hstack,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);
    int n_t = 1;
    int n_b = 1;
    int n_i = 1;
    while ( n_i > 0 )
    {
        partition<<<n_b,n_t>>>( d_d, d_l, d_h, n_i);
        int answer;
        cudaMemcpyFromSymbol(&answer, d_size, sizeof(int), 0,
cudaMemcpyDeviceToHost);
        if (answer < 1024)
          {
            n_t = answer;
          }
        else
          {
            n_t = 1024;
            n_b = answer/n_t + (answer%n_t==0?0:1);
          }
        n_i = answer;
        cudaMemcpy(arr, d_d,(h-l+1)*sizeof(int),cudaMemcpyDeviceToHost);
    }
}


int main()
{
    int arr[10] = {9,8,7,6,5,949,11,1,2,100};

    printf("Initial array: ");
    for(int i = 0; i < 10; i++){
        printf("%d " , arr[i]);
    }

    int n = sizeof( arr ) / sizeof( *arr );
    quickSortIterative( arr, 0, n - 1 );
```

Name : Manthan Gopal Dhole
Id : 201080020


```
    printf("\nApplying quick sort... \nDisplaying sorted array: ");

    for(int i = 0; i < 10; i++){
        printf("%d " , arr[i]);
    }
    return 0;
}
```

Output :

Initial array: 9 8 7 6 5 949 11 1 2 100
Applying quick sort...
Displaying sorted array: 1 2 5 6 7 8 9 11 100 949

```
Initial array: 9 8 7 6 5 949 11 1 2 100
Applying quick sort...
Displaying sorted array: 1 2 5 6 7 8 9 11 100 949
```

Explanation :

This code is an implementation of the quicksort algorithm using CUDA, a parallel computing platform and programming model developed by NVIDIA. The quicksort algorithm is used to sort an array of integers in ascending order.

The main function initializes an array of 10 integers and calls the quickSortIterative function to sort it. The quickSortIterative function takes three arguments: the array to be sorted, the lower index of the array, and the higher index of the array. It first initializes three arrays: lstack, hstack, and d_d, where lstack and hstack are used to store the lower and higher indexes of the partitions and d_d is a device (GPU) pointer that points to the array to be sorted in the device memory.

The function then starts a loop to iteratively sort the array using the quicksort algorithm. In each iteration, it calls the partition function which partitions the array based on a pivot

element and returns the lower and higher indexes of the two partitions. The partition function is executed in parallel on the device by multiple threads organized in blocks. The number of threads and blocks is determined dynamically based on the size of the current partition. The atomicAdd function is used to increment the d_size variable, which is shared by all threads, in order to dynamically determine the size of the next partitions.

After the partition function completes, the size of the next partitions is obtained by copying the value of d_size from the device to the host. If the size is less than or equal to 1024, the number of threads in the next partition is set to the size. Otherwise, the number of threads is set to 1024 and the number of blocks is calculated to cover the entire partition. The loop continues until all partitions have been sorted.

Finally, the sorted array is copied back to the host memory and printed.

Conclusion:

CUDA programming model is well-suited for hyper quicksort due to its ability to efficiently handle large amounts of data and its support for parallelism at the thread and block levels. The performance of hyper quicksort in parallel computing depends on various factors, including the size of the input array, the number of processing units available, and the quality of the load balancing scheme. Generally, hyper quicksort can achieve significant speedup and scalability when executed on a large number of processing units. However, care must be taken to ensure that the load is well-balanced across all the processing units to avoid performance degradation.