CONTENTS

INTRODUCTION:

Graphics Processing Units (GPUs) have emerged as powerful tools for accelerating computations, especially in parallelizable tasks like sorting and searching.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and API model created by **NVIDIA**. It allows developers to harness the computational power of NVIDIA GPUs for general-purpose processing, including sorting and searching algorithms.

Key Features:

Performance Profiling: Leveraging CUDA's parallel computing capabilities, users can conduct real-time performance analyses of algorithms. This facilitates insightful comparisons and elucidates algorithmic intricacies and optimisation strategies.

Scalability: With CUDA's distributed computing prowess, our platform seamlessly scales to tackle substantial datasets and intricate computational tasks.

Multi - Language/Platform Support: NVIDIA's Cuda API has multi-language support (Python:Pycuda, C:cuda.h,nvcc compiler ,etc) for running natively as well as over Cloud in Web Services like Google Colab.

Objectives:

- Implement few sorting algorithms .
- Implement few searching algorithms .

- Evaluate the performance of CUDA implementations compared to CPU implementations.
- Analyze the scalability of CUDA implementations with increasing input sizes by customising grid and block sizes.

CPU (Central Processing Unit):

- Designed for general-purpose computation.
- Typically has fewer cores (e.g., 4 to 64 cores).
- Optimized for low-latency task execution.
- Well-suited for tasks requiring complex decision-making and branching logic.

Simple "Hello World" Code in CPU:

```
int main( void ) {
printf( "Hello, World!\n" ); return 0;}
```

GPU (Graphics Processing Unit):

- Originally designed for rendering graphics but can be used for general-purpose computation.
- Has a large number of cores (e.g., hundreds to thousands of cores).
- Optimized for parallel processing, ideal for tasks with high data parallelism.
- Well-suited for tasks requiring high-throughput computation on large datasets.

”Hello World” in GPU:

```
#include <iostream>
_global_ void kernel( void ) {
printf( "Hello, World!\n" );}

int main( void ) {
kernel<<<1,1>>>();
return 0; }
```

CUDA C adds the __global__ qualifier to standard C. This mechanism alerts the compiler that a function should be compiled to run on a device instead of the host. In this simple example, nvcc gives the function *kernel()* to the compiler that handles device code, and it feeds *main()* to the host compiler as it did in the previous example.

GPU Assembly level code

The assembly language for GPUs varies depending on the GPU architecture and the programming model used. NVIDIA GPUs, for example, use an assembly-like language called PTX (Parallel Thread Execution) for CUDA programming. PTX is a low-level, virtualized assembly language that is compiled by the CUDA compiler into the actual machine code that runs on the GPU.

Here's an example of basic operations in PTX assembly code:

```
assembly
.version 6.5
.target sm_61
.address_size 64

// Kernel definition
.global .func  void add_kernel(
    .param .u64 add_kernel_param_0,
    .param .u64 add_kernel_param_1,
    .param .u64 add_kernel_param_2
)
{
    .reg .u64  %rd<3>;
    .reg .u32  %r<4>;
    .reg .u64  %rd1;
    .reg .pred %p<2>;

    // Load parameters into registers
    ld.param.u64    %rd1, [add_kernel_param_0];
    ld.param.u64    %rd2, [add_kernel_param_1];

    // Add the values
    add.u64         %rd3, %rd1, %rd2;
```

```
    // Store the result back to global memory
    st.global.u64   [%rd2], %rd3;

    ret;
}
```

In this example, the PTX code defines a kernel add_kernel that adds two 64-bit integers (add_kernel_param_0 and add_kernel_param_1) and stores the result in global memory (add_kernel_param_2).

To run this PTX assembly code, you would typically follow these steps:

1. **Write the PTX code:** Save the PTX code in a file with a .ptx extension.

2. **Compile the PTX code:** Use the NVIDIA PTX assembler (ptxas) to compile the PTX code into a CUDA object file. For example:

ptxas -o add_kernel.o add_kernel.ptx

3. **Link the CUDA object file:** Link the CUDA object file with your CUDA application code using the NVIDIA CUDA compiler (nvcc):

nvcc -o my_program my_program.cu add_kernel.o

4. **Run the CUDA application:** Execute the compiled CUDA application, which will run the PTX kernel on the GPU.

Note that these steps assume you are working with CUDA and an NVIDIA GPU. The exact steps may vary depending on your specific environment and toolchain.

CUDA PROGRAMMING:

CUDA provides a set of functions and syntax for programming NVIDIA GPUs. Here's an overview of some key concepts:

Kernel Function: A function that runs on the GPU. It's executed in parallel by multiple threads. Defined using _global_ keyword.

Kernel Call: Launching a kernel function from the CPU code. Done using <<<...>>> syntax.

Threads: Individual units of execution within a GPU. Thousands of threads can run in parallel.

Blocks: Threads are organized into blocks for better management. Blocks can contain multiple threads.

Grid: Collection of blocks. A kernel is executed as a grid of blocks.

CUDA MEMORY MANAGEMENT:

A. *cudaMalloc:* Allocates memory on the GPU.

B. *cudaMemcpy:* Copies data between CPU and GPU memory.

C. *cudaFree:* Frees memory allocated on the GPU.

Thread Blocks: The GPU organizes threads into groups called "thread blocks." Each thread block can contain a maximum number of threads, which depends on the GPU architecture. the number of threads per block and the maximum number of threads per block can vary between different GPU models and architectures. For example, for devices with a compute capability of 2.x, the maximum number of threads per block is 1024.

Grids: Threads blocks are organized into a grid. The grid is the collection of all thread blocks launched to execute the kernel.

Thread Indexing: Inside a kernel, each thread has a unique index that identifies its position within the grid and block structure. These indices are used to calculate which data element each thread should operate on.

Since the maximum number of Threads differ from one GPU to other, we need to know the configurations and architecture of the device one is working with.

CUDA DEVICE SPECIFIC PROPERTIES:

There can be more than one cuda capable devices which are interfaced together. To count the number of devices , we use cudaGetDeviceCount().

```
int count;
HANDLE_ERROR( cudaGetDeviceCount( &count ) );
```

After calling the above function , we can iterate through the device properties and query about relevant info. All these properties are stored in a C structure of type cudaDeviceProp.

Example structure :

```
struct cudaDeviceProp {
char name[256];
size_t totalGlobalMem;
size_t sharedMemPerBlock;
int regsPerBlock;
int warpSize;
size_t memPitch;
int maxThreadsPerBlock;
int maxThreadsDim[3];
int maxGridSize[3];
size_t totalConstMem;
int major;
int minor;
int clockRate;
size_t textureAlignment;
int deviceOverlap;
int multiProcessorCount;
int kernelExecTimeoutEnabled; int integrated;
int canMapHostMemory;
int computeMode;
int maxTexture1D;
int maxTexture2D[2];
```

```
int maxTexture3D[3];
int maxTexture2DArray[3];
int concurrentKernels;
}
```

DESCRIPTION OF DEVICE PROPERTIES

Table 3.1 CUDA Device Properties

DEVICE PROPERTY	DESCRIPTION
`char name[256];`	An ASCII string identifying the device (e.g., `"GeForce GTX 280"`)
`size_t totalGlobalMem`	The amount of global memory on the device in bytes
`size_t sharedMemPerBlock`	The maximum amount of shared memory a single block may use in bytes
`int regsPerBlock`	The number of 32-bit registers available per block
`int warpSize`	The number of threads in a warp
`size_t memPitch`	The maximum pitch allowed for memory copies in bytes

DEVICE PROPERTY	DESCRIPTION
`int maxThreadsPerBlock`	The maximum number of threads that a block may contain
`int maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
`int maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
`size_t totalConstMem`	The amount of available constant memory
`int major`	The major revision of the device's compute capability
`int minor`	The minor revision of the device's compute capability
`size_t textureAlignment`	The device's requirement for texture alignment
`int deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
`int multiProcessorCount`	The number of multiprocessors on the device
`int kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
`int integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
`int canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space
`int computeMode`	A value representing the device's computing mode: default, exclusive, or prohibited
`int maxTexture1D`	The maximum size supported for 1D textures

DEVICE PROPERTY	DESCRIPTION
`int maxTexture2D[2]`	The maximum dimensions supported for 2D textures
`int maxTexture3D[3]`	The maximum dimensions supported for 3D textures
`int maxTexture2DArray[3]`	The maximum dimensions supported for 2D texture arrays
`int concurrentKernels`	A boolean value representing whether the device supports executing multiple kernels within the same context simultaneously

```
Device 0: "NVIDIA GeForce MX250"
  CUDA Driver Version / Runtime Version          12.4 / 12.3
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 2048 MBytes (2147352576 bytes)
  (003) Multiprocessors, (128) CUDA Cores/MP:    384 CUDA Cores
  GPU Max Clock rate:                            1582 MHz (1.58 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.4, CUDA Runtime Version = 12.3, NumDevs = 1
Result = PASS
```

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices at the present depth before moving on to the vertices at the next depth level. A parallel approach to BFS can

significantly accelerate its execution by exploiting the parallelism inherent in the algorithm.

In a parallel BFS, multiple vertices at the same depth level can be explored concurrently, leading to a more efficient exploration of the graph. Here's a basic outline of BFS in parallel execution:

1. **Initialization:** Initially, all vertices are marked as unvisited. The starting vertex is marked as visited and added to a queue (or a frontier).

2. **Parallel Exploration:** In each iteration, the algorithm selects a subset of vertices from the queue (or frontier) to explore in parallel. Each thread in the parallel environment processes one vertex from this subset.

3. **Vertex Expansion:** For each vertex being processed, the algorithm examines all its neighboring vertices that have not been visited yet. If a neighboring vertex is found, it is marked as visited and added to the queue (or frontier) for further exploration.

4. **Synchronization:** After processing the vertices in the current depth level, the algorithm synchronizes to ensure that all threads have completed their tasks before moving on to the next depth level.

5. **Termination:** The algorithm terminates when there are no more vertices left to explore.

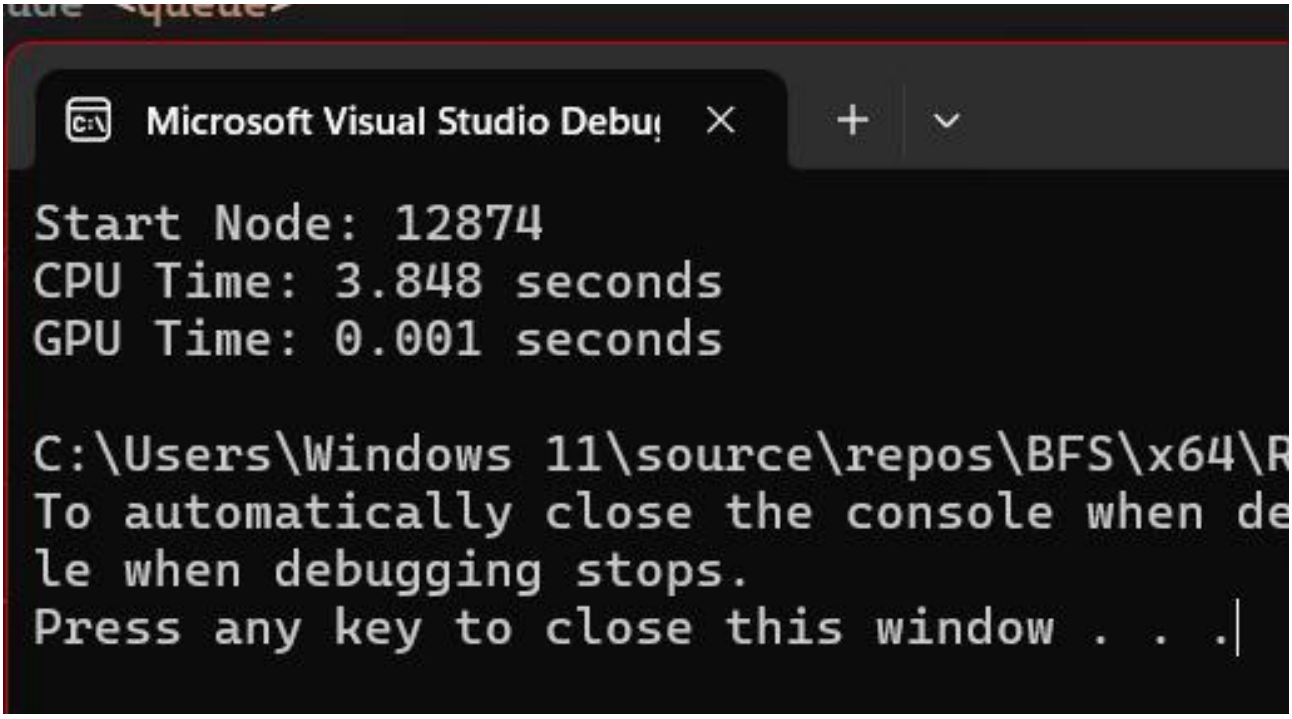
6. **Optimizations:** Various optimizations can be applied to improve the efficiency of the parallel BFS, such as using a work-stealing mechanism to balance the workload among threads, avoiding redundant exploration of already visited vertices, and utilizing efficient data structures for the frontier (e.g., a priority queue).

Algorithm:

```
// Kernel function for BFS on GPU
__global__ void BFS_GPU(int* adj_matrix, bool* visited, int* queue, int* queue_end, int* result, int start_node) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid == 0) {
        visited[start_node] = true;
        queue[0] = start_node;
        *queue_end = 1;
    }

    __syncthreads();

    while (*queue_end > 0) {
        int idx = atomicSub(queue_end, 1) - 1;
        int node = queue[idx];

        if (node == -1)
```

```
            break;

        for (int i = 0; i < N; ++i) {
            if (adj_matrix[node * N + i] && !visited[i]) {
                visited[i] = true;
                int new_idx = atomicAdd(queue_end, 1);
                queue[new_idx] = i;
            }
        }
    }

    if (tid == 0) {
        *result = 1;
    }
}
```

OUTPUT:



Parallel matrix multiplication can be significantly accelerated using GPUs due to their highly parallel architecture. The basic idea is to distribute the work of multiplying elements of two matrices across multiple threads or cores, allowing for simultaneous computation of multiple elements. All matrix operations are the most used when it comes to day to day application of GPUs (Artificial Intelligence and Machine Learning).

Here's a general outline of a parallel approach to matrix multiplication:

1. **Data Partitioning:** Divide the input matrices into smaller submatrices and distribute these submatrices across the available threads or cores. Each thread will be responsible for computing a portion of the final result.

2. **Compute Multiplication:** Each thread computes the multiplication of its assigned submatrices. For example, if thread (i, j) is responsible for computing element (i, j) of the result matrix, it will compute the dot product of row i of the first matrix with column j of the second matrix.

3. **Summation:** After computing the partial results, the individual results need to be summed up to obtain the final result. This can be done using a reduction operation, where each thread contributes its partial result to a shared variable that accumulates the final sum.

4. **Synchronization:** Ensure that all threads have completed their computations before proceeding to the summation step. This can be achieved using synchronization primitives provided by the parallel programming framework (e.g., CUDA or OpenCL).

5. **Optimizations:** Various optimizations can be applied to improve the efficiency of parallel matrix multiplication, such as using shared memory to reduce memory access times, minimizing thread divergence, and exploiting data locality to reduce memory bandwidth requirements.

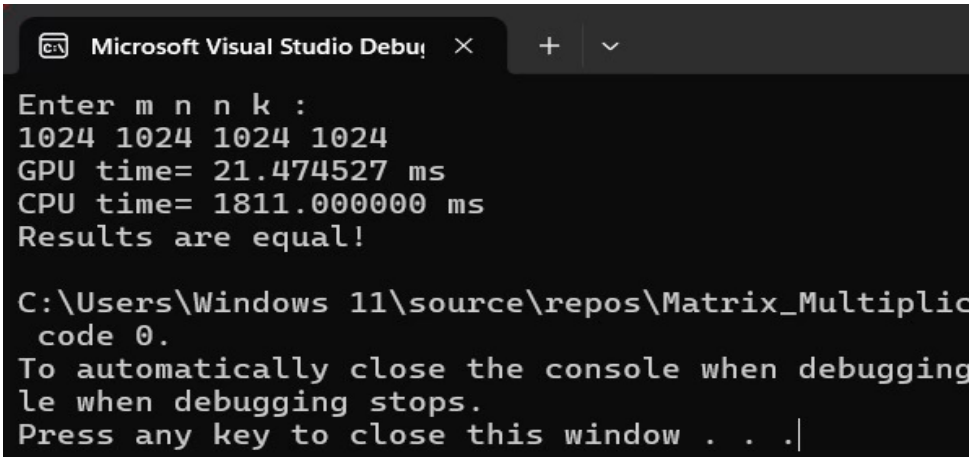
Algorithm:

```
// Kernel function for matrix multiplication on GPU
__global__ void multiply(float* left, float* right, float* res, int dim) {
    // Shared memory for tiles of left and right matrices
    __shared__ float Left_shared_t[BLOCK_SIZE][BLOCK_SIZE];
```

```
    __shared__ float Right_shared_t[BLOCK_SIZE][BLOCK_SIZE];

    // Compute indices for accessing data
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float temp = 0;

    // Loop over tiles of left and right matrices
    for (int tileNUM = 0; tileNUM < gridDim.x; tileNUM++) {
        // Load tiles into shared memory
        int j = tileNUM * BLOCK_SIZE + threadIdx.x;
        int i = tileNUM * BLOCK_SIZE + threadIdx.y;
        Left_shared_t[threadIdx.y][threadIdx.x] = left[row * dim + j];
        Right_shared_t[threadIdx.y][threadIdx.x] = right[i * dim + col];
        __syncthreads();

        // Accumulate result for this tile
        for (int k = 0; k < BLOCK_SIZE; k++) {
            temp += Left_shared_t[threadIdx.y][k] *
Right_shared_t[k][threadIdx.x];
        }
        __syncthreads();
    }
```

```
    // Store result to global memory
    res[row * dim + col] = temp;
}
```

OUTPUT:



Bitonic sort is a parallel sorting algorithm that can take advantage of the parallel processing capabilities of GPUs. It is particularly efficient for sorting arrays whose size is a power of two. The basic idea behind Bitonic sort is to first sort the input into a "bitonic sequence" (a sequence that first increases and then decreases), and then repeatedly merge adjacent bitonic sequences to form larger sorted sequences, until the entire array is sorted.

Here's how Bitonic sort is executed in parallel:

1. **Data Partitioning:** Divide the input array into smaller subarrays and distribute these subarrays across the available threads or cores. Each thread will be responsible for sorting a portion of the input array.

2. **Bitonic Sorting Network:** Implement a bitonic sorting network within each thread. A bitonic sorting network is a sequence of comparison and exchange operations that transforms a random sequence of elements into a bitonic sequence. The key property of a bitonic sorting network is that it can be implemented efficiently in parallel.

3. **Local Sorting:** Each thread applies the bitonic sorting network to its assigned subarray, resulting in a locally sorted subarray.

4. **Global Merge:** After the local sorting step, the array is in a "bitonic" state, where it first increases and then decreases. To sort the entire array, a series of merge operations are performed. These merge operations are typically implemented using parallel reduction and permutation operations.

5. **Synchronization:** Ensure that all threads have completed their sorting and merging operations before proceeding to the next step. This can be achieved using synchronization primitives provided by the parallel programming framework (e.g., CUDA or OpenCL).

6. **Repeat Merging:** Repeat the merging step until the entire array is sorted.

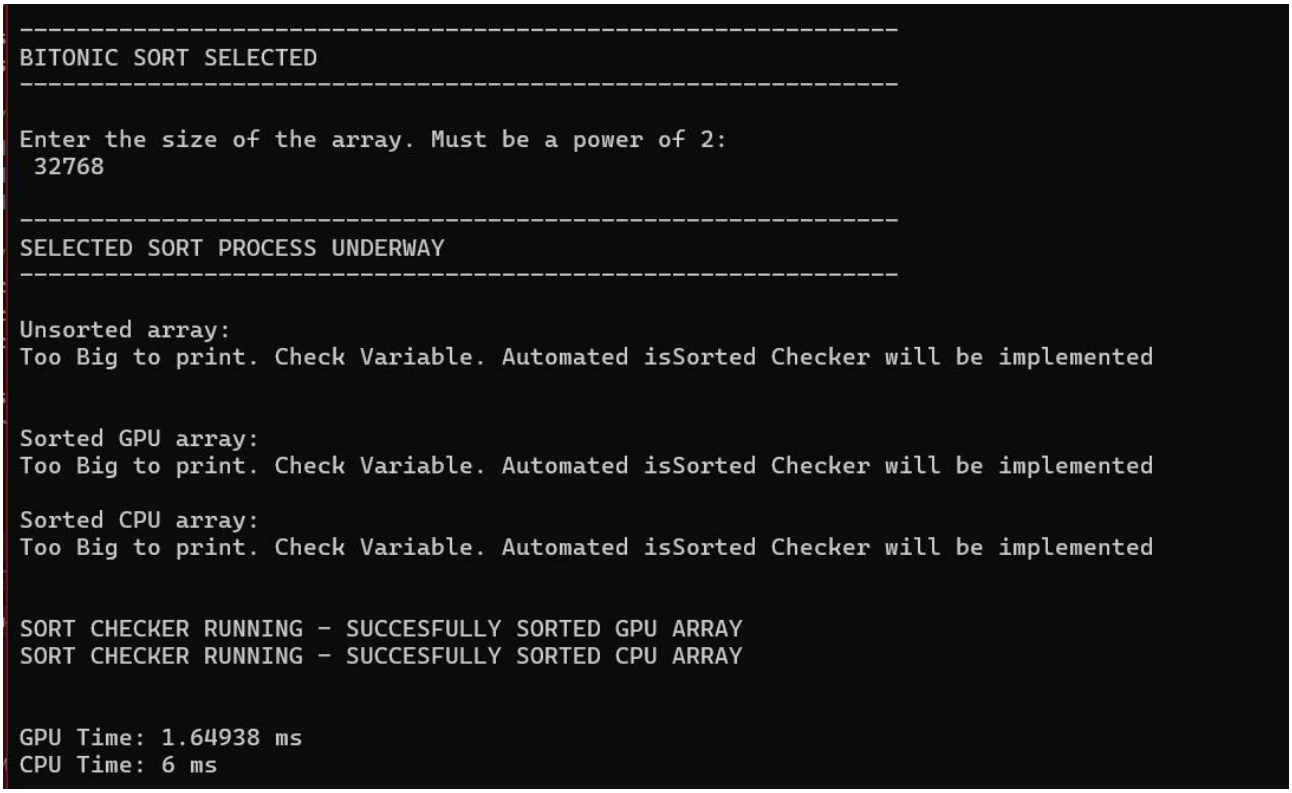
Algorithm:

```
//GPU Kernel Implementation of Bitonic Sort
__global__ void bitonicSortGPU(int* arr, int j, int k)
{
    unsigned int i, ij;

    i = threadIdx.x + blockDim.x * blockIdx.x;

    ij = i ^ j;
```

```
    if (ij > i)
    {
        if ((i & k) == 0)
        {
            if (arr[i] > arr[ij])
            {
                int temp = arr[i];
                arr[i] = arr[ij];
                arr[ij] = temp;
            }
        }
        else
        {
            if (arr[i] < arr[ij])
            {
                int temp = arr[i];
                arr[i] = arr[ij];
                arr[ij] = temp;
            }
        }
    }
}


//GPU Kernel for Merge Sort
__global__ void MergeSortGPU(int* arr, int* temp, int n, int width)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int left = tid * width;
    int middle = left + width / 2;
    int right = left + width;

    if (left < n && middle < n)
    {
        Merge(arr, temp, left, middle, right);
    }
}


//CPU Implementation of Merge Sort
void mergeSortCPU(int* arr, int* temp, int left, int right)
{
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;

    mergeSortCPU(arr, temp, left, mid);
    mergeSortCPU(arr, temp, mid + 1, right);

    merge(arr, temp, left, mid, right);
}

// Bitonic Sort for CPU
void bitonicSortCPU(int* arr, int n)
{
    for (int k = 2; k <= n; k *= 2)
    {
        for (int j = k / 2; j > 0; j /= 2)
        {
            for (int i = 0; i < n; i++)
```

```
            {
                int ij = i ^ j;

                if (ij > i)
                {
                    if ((i & k) == 0)
                    {
                        if (arr[i] > arr[ij])
                        {
                            int temp = arr[i];
                            arr[i] = arr[ij];
                            arr[ij] = temp;
                        }
                    }
                    else
                    {
                        if (arr[i] < arr[ij])
                        {
                            int temp = arr[i];
                            arr[i] = arr[ij];
                            arr[ij] = temp;
                        }
                    }
                }
            }
        }
    }
}
```

OUTPUT:



Radix sort works by sorting elements based on individual digits or bytes, starting from the least significant digit (or byte) and moving towards the most significant digit (or byte). It uses a stable sorting algorithm (such as counting sort) at each step to sort the elements based on the current digit or byte.

Here's how Radix sort is executed in parallel on a GPU:

1. **Data Partitioning**: Divide the input array into smaller subarrays and distribute these subarrays across the available threads or cores. Each thread will be responsible for sorting a portion of the input array.

2. **Digit Extraction:** For each element in the input array, extract the digit or byte at the current radix position. This can be done using bitwise operations or other methods, depending on the size of the radix.

3. **Counting Sort:** Perform a counting sort on the input array based on the extracted digits or bytes. Counting sort is a stable sorting algorithm that can be implemented efficiently in parallel.

4. **Global Merge:** After the counting sort step, the array is partially sorted based on the current radix position. To sort the entire array, a series of merge operations are performed. These merge operations are typically implemented using parallel reduction and permutation operations.

5. **Repeat for Each Radix:** Repeat steps 2-4 for each radix position, starting from the least significant digit or byte and moving towards the most significant digit or byte.

6. **Synchronization:** Ensure that all threads have completed their sorting and merging operations before proceeding to the next radix position. This can be achieved using synchronization primitives provided by the parallel programming framework (e.g., CUDA or OpenCL).

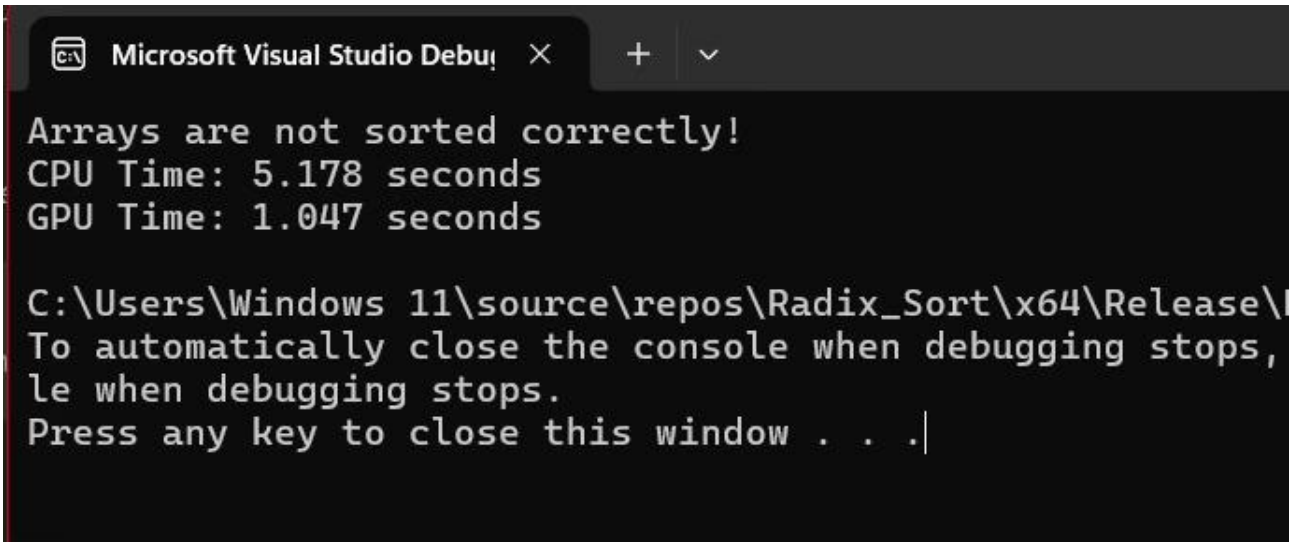
7. **Repeat Merging:** Repeat the merging step until the entire array is sorted.

Algorithm:

```
function findMax(array, size)
    max = 0
    for each element in array
        if element > max
            max = element
    return max

function countingSort(input, output, digit, size)
    initialize count array to store occurrences of each digit
    for each element in input
        d = (element / digit) % 10
        increment count[d]
    perform exclusive scan on count array
    for each element in input
        d = (element / digit) % 10
        index = scan[d]
        output[index] = element
        increment scan[d]

function radixSortGPU(array, size)
    allocate memory for input and output arrays on device
    copy array from host to device
    max = findMax(array, size)
```

```
    digit = 1
    while max / digit > 0
        countingSort<<<gridSize, blockSize>>>(input, output, digit,
size)
        copy output array back to input array on device
        digit *= 10
```

OUTPUT:



Advanced Implementations:

- **Advanced Algorithms Integration:** Expand the platform to cover advanced algorithms like graph algorithms and dynamic programming. Optimize these algorithms for parallel processing using Cuda, enabling users to efficiently tackle complex computational problems.

- **Machine Learning Integration:** Incorporate machine learning and AI algorithms into the platform. Utilize Cuda's capabilities

to execute these algorithms on GPU-accelerated hardware, enhancing performance and scalability.

- **Visualization and Analytics:** Develop interactive visualization tools and analytics dashboards to provide users with deeper insights into algorithmic performance and behavior.

Conclusion:

The implementation and analysis of sorting and searching algorithms in CUDA have provided valuable insights into the potential of GPU acceleration for computational tasks. The parallel approach to algorithms such as Bitonic Sort, Radix Sort, Breadth First Search (BFS) has demonstrated significant performance improvements compared to their sequential counterparts.

The parallel implementations of sorting algorithms, including Bitonic Sort and Radix Sort, have showcased the ability of CUDA to exploit the parallel processing power of GPUs. These algorithms have shown substantial speedup, especially when dealing with large datasets. The parallel nature of these algorithms aligns well with the massively parallel architecture of GPUs, making them well-suited for GPU acceleration.

Furthermore, the parallel approach to BFS has also demonstrated the benefits of GPU acceleration for graph traversal and search tasks. The ability to explore multiple vertices simultaneously in BFS has led to faster execution times, particularly for large graphs.

In conclusion, the project has successfully implemented and analyzed various sorting and searching algorithms in CUDA, highlighting the

advantages of GPU acceleration for these computational tasks. The results obtained underscore the potential of CUDA and GPUs in accelerating a wide range of parallelizable algorithms, with implications for areas such as data processing, scientific computing, and machine learning.