SEARCHING

GLOBAL MEMORY: Searching

- The search we have two options: a binary search.
- A binary search takes advantage of the fact we have a sorted list of samples from the previous step.
- It works by dividing the list into two halves and asking whether the value it seeks is in the top or bottom half of the dataset.
- It then divides the list again and again until such time as it finds the value.
- The worst case sort time for a binary search is log2(N).

Parallel Searching

```
#include <iostream>
#include <cstdio>

// Kernel function for binary search
__global__ void binarySearchKernel(int *arr, int target, int left, int right, int *result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

Parallel Searching

```
while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            atomicExch(result, mid); // Store the result in a thread-safe manner
            return;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
}
```

Parallel Searching

```
int main() {
    const int arraySize = 1024;
    const int target = 42;

    int *hostArray, *deviceArray, *deviceResult;

    // Allocate memory on host and device
    hostArray = new int[arraySize];
    cudaMalloc(&deviceArray, arraySize * sizeof(int));
    cudaMalloc(&deviceResult, sizeof(int));
```

Parallel Searching

```
    // Initialize the sorted array on the host
    for (int i = 0; i < arraySize; ++i) {
        hostArray[i] = i * 2;
    }
// Copy data from host to device
    cudaMemcpy(deviceArray, hostArray, arraySize * sizeof(int),
cudaMemcpyHostToDevice);


    // Set up grid and block dimensions
    int threadsPerBlock = 256;
    int blocksPerGrid = (arraySize + threadsPerBlock - 1) /
threadsPerBlock;
```

Parallel Searching

```
 // Launch kernel
    binarySearchKernel<<<blocksPerGrid, threadsPerBlock>>>(deviceArray, target, 0, arraySize - 1, deviceResult);
// Copy result from device to host
    int result;
    cudaMemcpy(&result, deviceResult, sizeof(int), cudaMemcpyDeviceToHost);
if (result != -1) {
        std::cout << "Element " << target << " found at index " << result << std::endl;
    } else {
        std::cout << "Element " << target << " not found in the array." << std::endl;
    }
```

Parallel Searching

```
    // Clean up
    delete[] hostArray;
    cudaFree(deviceArray);
    cudaFree(deviceResult);
    return 0;
}
```