ASSIGNMENT #4

PROGRAMMING MASSIVELY PARALLEL PROCESSORS

Due Date:	As specified on the Moodle
Topics covered:	Radix Sort, shared memory, histogram, scan
Deliverables:	The submitted project files are kernel.cu. The files should be submitted according to the stipulations set out in the course syllabus.
Objectives:	Learn how to use shared memory, atomic operation, histogram and scan pattern to write more efficient CUDA program.

1 Background

Radix sort is a non-comparative sorting algorithm that processes keys digit by digit. For fixed-width integer keys, radix sort has time complexity $O(kN)$, where k is the number of digits (or bit groups).

On GPUs, radix sort can be efficiently parallelized by:

- Processing multiple elements concurrently
- Using parallel prefix sums (scan)
- Exploiting shared memory for local histograms
- Minimizing global memory accesses

In this assignment, you will implement a **least-significant-bit (LSB)** radix sort for 32-bit unsigned integers.

2 Problem Description

You are given an unsorted array of N unsigned 32-bit integers stored in GPU global memory. Your task is to sort the array in ascending order using a **parallel radix sort** implemented in CUDA.

2.1 Radix Configuration

You may assume:

- Keys are 32-bit unsigned integers
- Radix size: 2^r (e.g., $r = 1, 2$, or 4 bits per pass)
- Number of passes: $32/r$

2.2 Expectation

You are expected to

1. Understand radix sort's parallelization opportunities
2. Master CUDA memory hierarchy utilization
3. Analyze and optimize GPU kernel performance
4. Develop benchmarking and profiling skills
5. Compare algorithmic approaches for GPU architectures

3 Implementation Requirements

1. Implement the radix sort fully on the GPU using CUDA kernels.
2. Use parallel prefix sum (scan) to compute output positions.
3. Avoid using `thrust::sort` or other library sorting routines.
4. Your implementation must work for arbitrary N (not just powers of two).
5. Correctness must be verified against e.g. a CPU reference sort.

3.1 Suggested Algorithm Structure

Each radix pass may follow these steps:

1. Extract the current digit (or bit group).
2. Compute a histogram for each radix bucket.
3. Perform an exclusive prefix sum on the histogram.
4. Scatter elements into the correct output positions.
5. Swap input and output buffers for the next pass.

3.2 Different Phases

The implementation requirements are shown as follows:

1. Baseline Implementation
    - Implement a basic 32-bit radix sort with:
        - Digit-wise processing: 4-bit digits (8 passes)
        - Histogram computation: Count occurrences per digit per block
        - Prefix sum: Parallel scan implementation
        - Data reordering: Scatter elements to correct positions
    - Support: 0.1M to 10M elements, random uniform distribution
2. Enhancement:

  - Shared memory histograms: Reduce global memory traffic
  - Coalesced memory access: Ensure contiguous global memory patterns
  - Thread coarsening: each thread is assigned to multiple keys in the input list instead of just one
  - WARP level operation

We should analyze the performance metrics, with required measurements:

- Throughput: Elements sorted per second
- Speedup: Your GPU version vs. GPU reference baseline
- Kernel breakdown: Time per radix pass
- Memory bandwidth: Achieved vs. peak bandwidth
- Occupancy: Achieved SM occupancy

4 What to do

4.1 CUDA Kernel Overview

You are expected to write kernels similar to the following components:

- Block-level histogram kernel
- Prefix-sum (scan) kernel
- Scatter kernel

4.2 Experimental Evaluation

You must evaluate your implementation with different input sizes:

- $N = 10^5, 10^6, 10^7$
- Randomly generated keys

Report the following:

- Execution time (kernel-only and total time)
- Speedup compared to GPU reference baseline

4.3 Discussion

Discuss:

- Bottlenecks in your implementation
- Impact of radix size (r)
- Memory access patterns and coalescing
- Synchronization overhead

4.4 What you have to do

Write CUDA version of three kernel functions and one host function in kernel.cu (as shown below and in the template).

```
// Step 1:  Compute block histograms
__global__ void compute_histogram(
    unsigned int* d_in,
    unsigned int* d_block_hist,
    unsigned int shift,
    unsigned int n) {

}

// Step 2: GPU Global Prefix Sum
__global__ void compute_global_offsets(unsigned int* d_hist,
    unsigned int* d_block_offsets,
    unsigned int* d_digit_offsets,
    int num_blocks) {


}
// Step 3: scatter
__global__ void scatter(unsigned int* d_out,
    unsigned int* d_in,
    unsigned int* d_block_offsets,
    unsigned int* d_digit_offsets,
    unsigned int shift,
    unsigned int n) {

}

extern "C" void radix_sort_iteration(unsigned int* d_out,
    unsigned int* d_in,
    unsigned int* d_hist,
    unsigned int* d_block_offsets,
    unsigned int* d_digit_offsets,
    unsigned int shift,
    unsigned int n) {

}
```

Please note that *kernel_base.cu* is the baseline for benchmark test. It is provided in the VPL on the Moodle. Its source code is not provided in the template. The host function in *kernel_base.cu* can be used as the reference for your submission.

5 Grading Guideline

Please ensure that:

1. Functional Correctness: Produces correct result (CUDA C/C++ version is correct).
2. Coding:
   (a) Correct usage of CUDA library calls and C extensions.
   (b) Correct usage of thread id's in computation.
3. Pass the VPL test cases.

Your submission will be graded according to the test results. The list above is non-exhaustive.

The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.