CIS6930/4930 Summer 2017, Project 3

This is an open project to be finished by groups of up to **three** students. You can propose your own topic or choose one from the topics I recommend (see the bottom of this page for details). These topics listed also give you some ideas on the magnitude and flavor of the projects I wish to see. Basically, you are expected to implement a non-trivial data structure or algorithm (either as a standalone program or one integrated into an existing software system) using CUDA. For example, you can implement parallel search algorithm against a tree-based index in a database management system such as PostgreSQL.

Project requirements

1. Your code should be clearly documented and commented. Code without detailed comments are generally regarded as of little value to people who takes over the project or simply wants to understand your design. You are required to submit all the relevant code and accompanying documents.
2. I expect you to write a report in the format of a publishable paper. In the report, you should clearly state the objective of your project, your design of the data structure or algorithm, and relevant testcases by which the grader can evaluate your code.
3. Your project will be graded according to the functionality you implemented and the details and format of your report. A software demo is expected during the last week of class. The report and code should be submitted via Canvas by 11:30pm, July 23, 2017 (Sunday).

This is a group project. Once you form a group, please choose a topic to work on and inform the instructor of the topic. I also recommend you choose one person as the team leader to represent your group in communicating with the instructor and coordinate your activities inside the group.

The following are a few sample topics that you can choose from. The following descriptions are very concise; please schedule an appointment with the instructor or the existing team leader for more details. This part of the document will be updated with more project descriptions.

1. **Parallel hash join in Data Streams**. See the following pages of this document for details.
2. **Radix sort** is a non-comparative sorting algorithm. It sorts the data by partitioning the keys into buckets where the selected digits of the keys are the same in each bucket. A histogram is built to count the number of keys in the buckets and then the keys are reorganized so that those in the same bucket are clustered together. There are two variants of the radix sort: MSD (Most Significant Digits) and LSD (Least Significant Digits). In this project, you are to implement a MSD radix sort algorithm on GPU which sorts the data starting from the most significant bits. Since the number of bits selected decides the number of buckets, you can sort a small number of bits each time and run multiple passes so that the histogram can be kept in memory. The input data should be 32-bit signed integers and your program should take arbitrary data size within the range of 32-bit signed integer. You are encouraged to think up

optimization techniques that take advantage of the memory hierarchy of the GPU and any software features of the CUDA.

3. **Segmented reduction** is a very common procedure in parallel computing. It computes the reductions of many irregular-length segments in a single input sequence. In this project, you are to implement a parallel segmented reduction on GPU. The program takes two input arrays: an array of input data sequence, an array of the starting position of the segments. It returns an array of the reductions for all the segments. All data type are 32-bit signed integers. The most important part of the algorithmic design is to balance the load among threads since the segments are of different length. You are encouraged to think up optimization techniques that take advantage of the memory hierarchy of the GPU and any software features of the CUDA.

4. **Nearest Neighbor Search** (NNs) is a form of search that finds the point in a given input set that is closest to a given point. The closeness is measured using the Euclidian distance between the points. A direct generalization of this problem is a *k*-NN search, where we need to find the *k* closest points. We are working in a system that a group of query may need to output their *k*-NN. You need to develop a parallel program to efficiently find the K-NN points for a set of input query. K is the input to the program. The input points can be organized in a tree data structure to reduce the number of comparisons; however, the efficiency of your code will be graded based on the efficiency of your designed K-NN updating algorithm. In this project, the tree data structure is largely implemented (you may need to modify it for your k-NN implementation) and you can focus on the updating algorithm. **Spatial joins** can also be implemented based on the aforementioned tree data structure.

Project Title: Parallel Hash Join in Data Streams

Background:
A data stream is an unbounded sequence of stream tuples. Stream join, also called *window join* in the literature, is a class of join algorithms for joining infinite data streams. Window join addresses the infinite nature of the data streams by joining stream data tuples that lie within a *sliding window* and that match a certain join condition. Hence, each tuple in a sliding window should be compared with all tuples existing in the opposite sliding window of the other data stream for the join predicate (see Figure 1). Since the typical rate of such comparisons is significantly large in data stream applications, there is a strong demand to parallelize the stream join on GPUs in order to improve the join operation throughput.
In this project, we focus on parallel computing of *hash-based* stream join (a more efficient algorithm than the nested loop join) on GPUs.

Project Description:
Symmetric hash joins (SHJ) is a special type of hash join well-suited for stream join scenarios since it supports the sliding window semantics by its incremental nature. SHJ for two input streams is briefly shown as following (see Figure 2)

- For each input stream create a hash table based on its join attribute
- For each new tuple arrived in an input stream
    - hash and insert into the stream hash table
    - Probe the opposite hash table for finding the matching tuples
    - Output the join results

In this project, we would like to design the parallel SHJ algorithm and implement it on GPU architecture.

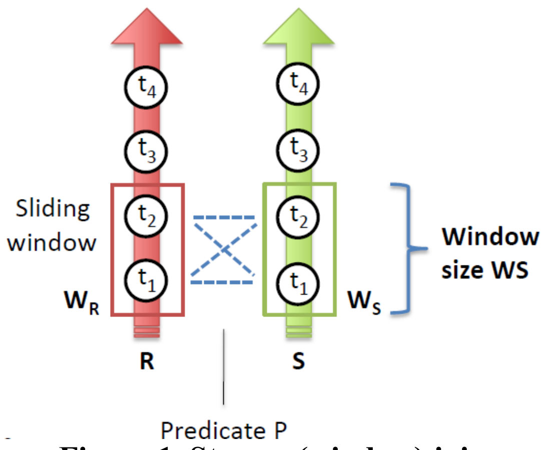

Figure 1. Stream (window) join

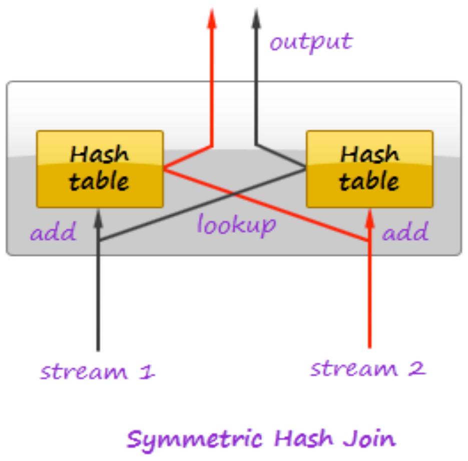

Figure 2. Symmetric Hash Join (SHJ)