CIS6930/4930 Summer 2016, 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 examples of 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 25, 2016 (Monday).

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. **Computing Support Vector Machines (SVM) using GPUs**. See the following pages of this document for details. More details can be obtained from our TA - Zhila Nouri (zhila@mail.usf.edu).

3. **SDH processing of large-scale data.** This can be seen as an extension of your Project 2. In particular, you will design memory management mechanisms for GPU global memory, implement CUDA kernels and evaluate the performance of such kernels. Your main task is to write the kernel to compute SDH with the assumption that input data are too large to place in global memory. The straightforward method to handle this is dividing data into small block. Then, load some of data block into GPUs memory and start the computation. However, the efficient way to load block of

data in to global memory is required for this project. Because loading data to global memory is a costly operation, you need to reduce the total amount of data transmitted as much as possible. To evaluate your kernel, you may use an analytical model that quantifies the number of access to global memory or empirically run the CUDA program and plot the running time of your kernel under different data sizes.

4. **Identification of Drug-Drug interaction (DDI).** DDI is a very important topic in clinical pharmacy and pharmaceutical research. DDIs are frequently reported in scientific and medical journals, making literature the most effective source for DDI detection. We believe certain fields of MEDLINE records could be used to extract implicit DDI and the underlying molecular mechanisms. Information from MEDLINE data can be efficiently analyzed by using computational methods. We have developed a three-stage algorithm to identify compounds that have interactions with drugs of interest by analyzing the *Substances* field in MEDLINE records. A CPU version of that implemented by the Python language needs tens of minutes to compute the results for a typical queried drug. This is unacceptable in an online service that may have to serve many concurrent queries. Since the samples are independent, parallel processing is obviously the strategy to follow to boost performance. In particular, we propose a parallel version of the algorithm geared towards running on modern Graphics Processing Units (GPUs).

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.

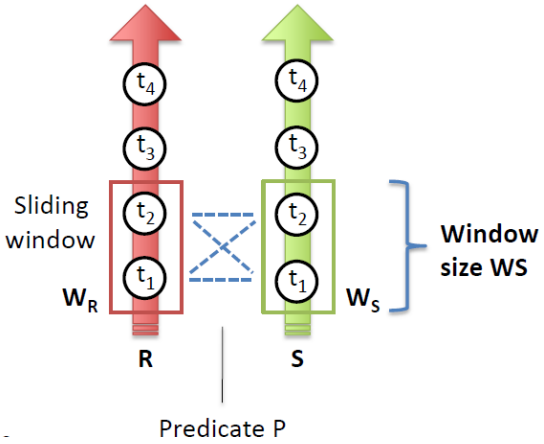


Figure 1. Stream (window) join

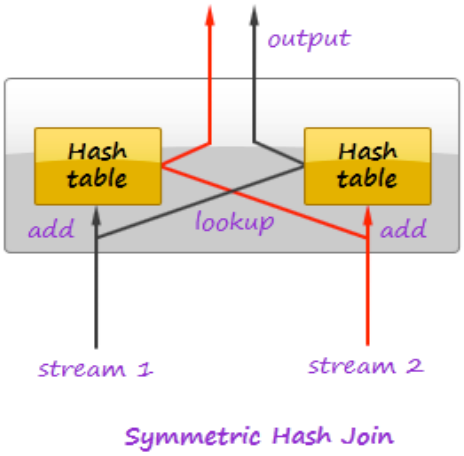
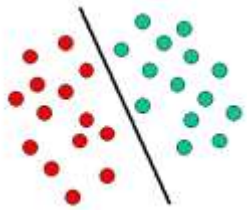


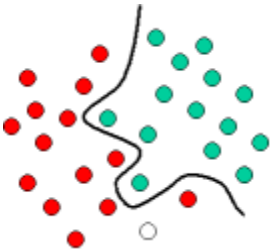
Figure 2. Symmetric Hash Join (SHJ)

Support Vector Machines (SVM)

Support Vector Machines are based on the concept of decision planes that define decision boundaries. A decision plane is one that separates between a set of objects having different class memberships. A schematic example is shown in the illustration below. In this example, the objects belong either to class GREEN or RED. The separating line defines a boundary on the right side of which all objects are GREEN and to the left of which all objects are RED. Any new object (white circle) falling to the right is labeled, i.e., classified, as GREEN (or classified as RED should it fall to the left of the separating line).

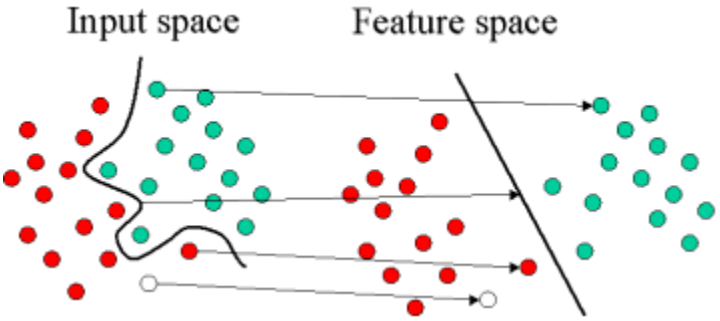


The above is a classic example of a linear classifier, i.e., a classifier that separates a set of objects into their respective groups (GREEN and RED in this case) with a line. Most classification tasks, however, are not that simple, and often more complex structures are needed in order to make an optimal separation, i.e., correctly classify new objects (test cases) on the basis of the examples that are available (train cases). This situation is depicted in the illustration below. Compared to the previous schematic, it is clear that a full separation of the GREEN and RED objects would require a curve (which is more complex than a line). Classification tasks based on drawing separating lines to distinguish between objects of different class memberships are known as hyperplane classifiers. Support Vector Machines are particularly suited to handle such tasks.



The illustration below shows the basic idea behind Support Vector Machines. Here we see the original objects (left side of the schematic) mapped, i.e., rearranged, using a set of mathematical functions, known as kernels. The process of rearranging the objects is known as mapping (transformation). Note that in this new setting, the mapped objects (right side of the schematic) is

linearly separable and, thus, instead of constructing the complex curve (left schematic), all we have to do is to find an optimal line that can separate the GREEN and the RED objects.



Sequential minimal optimization (SMO)

SMPO is an algorithm for solving the quadratic programming (QP) problem that arises during the training of support vector machines.

Consider a binary classification problem with a dataset (x1, y1), ..., (xn, yn), where xi is an input vector and yi ∈ {-1, +1} is a binary label corresponding to it. A soft-margin support vector machine is trained by solving a quadratic programming problem, which is expressed in the dual form as follows:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(x_i, x_j) \alpha_i \alpha_j,$$

subject to:

$$0 \leq \alpha_i \leq C, \quad \text{for } i = 1, 2, \ldots, n,$$

$$\sum_{i=1}^{n} y_i \alpha_i = 0$$

Where C is an SVM hyperparameter and K(xi, xj) is the kernel function, both supplied by the user; and the variables α_i are Lagrange multipliers.

SMO is an iterative algorithm for solving the optimization problem described above. SMO breaks this problem into a series of smallest possible sub-problems, which are then solved analytically. Because of the linear equality constraint involving the Lagrange multipliers α_i, the smallest possible problem involves two such multipliers. Then, for any two multipliers α_1 and α_2, the constraints are reduced to:

$$0 \leq \alpha_1, \alpha_2 \leq C,$$
$$y_1\alpha_1 + y_2\alpha_2 = k,$$

and this reduced problem can be solved analytically: one needs to find a minimum of a one-dimensional quadratic function. k is the negative of the sum over the rest of terms in the equality constraint, which is fixed in each iteration.

The algorithm proceeds as follows:

1. Find a Lagrange multiplier α_1 that violates the Karush–Kuhn–Tucker (KKT) conditions for the optimization problem.
2. Pick a second multiplier α_2 and optimize the pair (α_1, α_2).
3. Repeat steps 1 and 2 until convergence.

When all the Lagrange multipliers satisfy the KKT conditions (within a user-defined tolerance), the problem has been solved. Although this algorithm is guaranteed to converge, heuristics are used to choose the pair of multipliers so as to accelerate the rate of convergence.

Goal in this project:

Try to find a parallel optimized algorithm to run the SMO algorithm on GPU.