Astronomy and Computing 20 (2017) 155–159

Contents lists available at ScienceDirect

Astronomy and Computing

journal homepage: www.elsevier.com/locate/ascom

Full length article

Faster catalog matching on Graphics Processing Units☆

M.A. Lee [a,*], T. Budavári [b]

[a] Department of Computer Science, Johns Hopkins University, Baltimore, MD, 21218, USA
[b] Department of Applied Math and Statistics, Johns Hopkins University, Baltimore, MD, 21218, USA

ARTICLE INFO

Article history:
Received 27 February 2017
Accepted 2 August 2017
Available online 9 August 2017

Keywords:
Methods
Statistical
Astrometry
Catalogs
Surveys

ABSTRACT

One of the most fundamental problems in observational astronomy is the cross-identification of sources. Observations are made at different times in different wavelengths with separate instruments, resulting in a large set of independent observations. The scientific outcome is often limited by our ability to quickly perform associations across catalogs. The matching, however, is difficult scientifically, statistically as well as computationally. The former two require detailed physical modeling and advanced probabilistic concepts; the latter is due to the large volumes of data and the problem's combinatorial nature. In order to tackle the computational challenge and to prepare for future surveys we developed a new implementation on Graphics Processing Units. Our solution scales across multiple devices and can process hundreds of trillions of crossmatch candidates per second in a single machine.

1. Introduction

Modern telescopes produce vast volumes of data every night. With the current and upcoming advances in technology, survey data-sets are growing at a tremendous pace. To maximize the scientific value of each experiments we often need to combine their observations with other surveys. The identification of objects across multiple catalogs and surveys can lead to new discoveries and breakthroughs but the speed of matching is a limiting factor when combining the large outputs of many experiments. A number of studies have focused on the statistical aspect of this challenge (Budavári and Szalay, 2008; Heinis et al., 2009; Kerekes et al., 2010; Budavári and Loredo, 2015) but they all rely on fast 2-way matching engines to find candidate associations first.

The current solutions including the SkyQuery (Dobos et al., 2012; Budavári et al., 2013) and the CDS X-Match Service (Boch et al., 2016) use hierarchical indexes and space-filling curves to accelerate the process (Kunszt et al., 2001; Górski et al., 2005). While their performance is great they are far from the speed of interactive data exploration, the ability to do on-the-fly catalog federation would be a game-changer. To illustrate the big-data aspect of the project, let us consider a relatively small scenario of matching GALEX (50 million objects) with SDSS DR7 (150 million objects), a naïve implementation would require 7.5 quadrillion (10^{15}) comparisons.

In this paper we describe a novel approach that takes advantage of the extreme parallelism available on modern GPUs as well as an efficient method for reducing the total number of comparisons. The tool we present can crossmatch at rates of over a trillion candidate pairs per millisecond. We will limit our investigation to matching only two catalogs but without assuming that they are sorted or indexed in any way ahead of time. This choice is motivated by the fact that n-way associations can be built up by 2-way matching using the best guess direction of the partial matches (Budavári and Szalay, 2008).

2. Divide and conquer

The combinatorial scaling of a naïve matching approach can be remedied by quickly eliminating pairs at large separations. This is often achieved by partitioning the sky and considering only nearby areas instead of the entire sphere. These heuristic algorithms often rely on hierarchical division schemes such as Igloo (Crittenden, 2000), Hierarchical Triangular Mesh (HTM; Kunszt et al., 2001), HEALPix (Górski et al., 2005) or SDSSPix (Scranton et al.).

2.1. Building on the zones algorithm

Our approach follows the division scheme of the *Zones Algorithm* of Gray et al. (2007), which employs a much simpler scheme. It breaks up the sky into constant declination rings called the *zones*; see Fig. 1. All zones have the same *height*, h, measured in angle. For details on choosing a good value for h, see Section 3. A zone

☆ This code is registered at the ASCL with the code entry ascl:1303.021.

* Corresponding author.
E-mail address: matthiaslee@jhu.edu (M.A. Lee).

http://dx.doi.org/10.1016/j.ascom.2017.08.001
2213-1337/

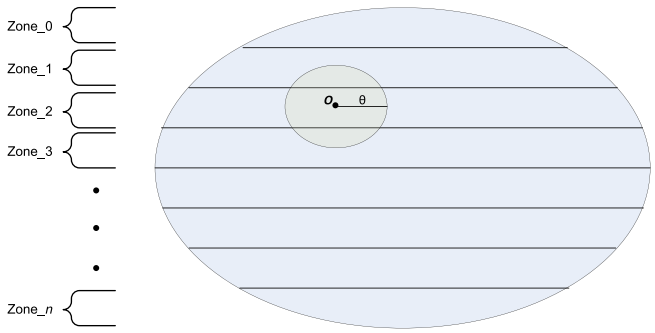

Fig. 1. The Zones Algorithm: the coordinate plane is subdivided along one axis into small strips of height h called *zones*. To find an object within a search radius, θ, all zones are searched which overlap with search radius.

identifier is assigned to each source i based on its declination $\delta_i \in [-90°, 90°]$ as given by

$$Z_i = \left\lfloor \frac{\delta_i + 90°}{h} \right\rfloor , \tag{1}$$

where $\lfloor\ \rfloor$ indicates the floor function rounding down to the closest integer.

Sources with the same values are in the same zones, which creates easy and computationally cheap division of sources across catalogs. The simplicity of this equation enables the immediate implementation in any language unlike the more complicated schemes listed above. This also fits well with the indexing facilities of relational database management systems (RDBMS). For example, the SkyQuery solution relies on zones for the largest matching problems, as an optimal query plan can essentially stream through the data on the hard drive with high efficiency and little memory overhead. The key for crossmatching is to use the same zone layout across all catalogs and eliminate all zone pairs that are farther than a specified search radius. The resulting set of zone pairs effectively mitigates any overlap needed to account for positional errors. Within the zones it can be beneficial to further sort by the right-ascension of the objects to quickly eliminate candidates that are too far apart even before calculating the separation between sources.

2.2. Layers of parallelism

Modern GPUs can run thousands of threads in parallel. In addition they also have superior memory bandwidth as compared to CPUs. The capacity of RAM, however, is somewhat more limited than that of today's servers. With these parameters in mind, we design our architecture to take maximum advantage of multiple GPUs in a single box. We will further assume that one of the catalogs, the smaller, can fit in the computer's memory, which will increase the speed of the execution. This is not, however, a significant constraint considering that 1 billion sources can be stored in 24GB of memory with 8-byte numbers for object identifier and the two celestial coordinates.

We slice the input catalogs into suitable *segments* that fit on the GPUs and build a job scheduler to process pairs of these segments from two catalogs. For the purposes of these next sections, let us consider two unsorted catalogs, `Catalog_A` and `Catalog_B`, each catalog consisting of objects identified by their *ObjId* (64bit integer), *RA* (double) and *Dec* (double). Our method breaks down into two main levels of parallelism. At the high level we distribute the problem across GPUs and at the lower level we parallelize across the many-core architecture within each GPU.

Input : Catalogs *A* and *B*, each containing objects identified by (id, ra, dec) and (id', ra', dec'), respectively

gpu-foreach *Segment of A* **do**
  Load Segment from disk;
  Copy to GPU;
  Sort by Zones and RA;
  Identify Zone boundaries;
  Copy back from GPU;
end

while *Segments in B remain* **do**
  gpu-foreach *Segment of B per GPU* **do**
    Load Segment from disk;
    Copy to GPU;
    Sort by Zones and RA;
    Identify Zone boundaries;
    Copy back from GPU;
  end
  gpu-foreach *Segment–Segment pair* **do**
    Compute distance metric for every object within each Zone–Zone pairing;
    Accumulate comparison results;
  end
end

Output: List of Object–Object pairings (id, id'), where (ra, dec) and (ra', dec') are considered a match

Algorithm 1: High-level Crossmatch Processing Steps

2.2.1. Preparing the segments

At runtime we begin the loading process of each catalog by subdividing each into multiple segments of size n. The size is chosen such that two segments, plus the overhead needed for processing, can fit into GPU memory. While GPU memory itself is very fast, transfers to and from are comparatively slow and hence should be kept to a minimum. The division of the catalogs into segments is purely a construct allowing us to manage the data more effectively in order to fit the data onto the GPU and minimize GPU-memory transfer overhead.

We begin by loading the smaller of the two input catalogs, into CPU memory, segmenting it as we go. After each segment is read from disk, it is loaded onto a GPU and sorted by the zone identifiers Z and right-ascension using the C/C++ CUDA *Thrust* library (Bell and Hoberock, 2011). This is done via a custom comparator implemented as a functor which on-the-fly and in parallel calculates the Z values. Arithmetics on the GPU are very fast and repeated calculations of the zone identifier do not slow down the process as we save on memory transfer, which is the typical bottleneck. Sorting by zone identifiers is only part of the battle, we also need to identify the zone boundaries, enabling us to easily separate out zones at execution time. This has also been implemented in parallel using the `thurst :: lower_bound()` and `thrust :: upper_bound()` search functions, which are based on Thrust's vectorized binary search.

We then loop over the larger catalog, reading one segment per available GPU. These segments are also loaded onto the GPUs and sorted by zones. At this point the preparation has taken place and the system is ready to loop through the segments of the catalog loaded in CPU memory.

2.2.2. Jobs and workers

A dynamic execution environment is implemented where a pool of worker threads, one thread per GPU, wait for jobs consisting of two segments, one from each input catalog. A *job manager* keeps