Fast Cross-matching of Astronomical Catalogs on GPUs

Institute for Data Intensive Engineering and Sciences
Johns Hopkins University

Matthias A Lee & Tamás Budavári
The Johns Hopkins University

Introduction

With the advance of technology, modern sensors have become cheaper, smaller and more effective. Proportionately the amount of data collected grows exponentially. This growth is especially true in the field of Astronomy where modern telescopes can collect millions/billions of data-points in a single day. Observations are made in different wave lengths, at different times and from different locations and instruments, resulting in a large set of independent observations with variations based on *what* was observed by *which* instrument at *what* point in time. Figure 1 depicts a galaxy in 5 different wavelengths, observed by 5 different surveys. One can immediately see the difference in intensity in different areas of the snapshots. For example observe the two high intensity spots off to the left in the VLA observation, now compare the same regions with the other surveys.

Figure 1: An example of the same Galaxy in 5 different surveys and wavelengths

Cross-referencing or cross-identifying common objects between multiple surveys can lead to the new discoveries and breakthroughs as each survey contributes its own unique datapoints.
The problem is correctly cross-identifying the same objects across all different observations, taking into account all factors.
Due to the growth in size of these datasets it has become increasingly computationally difficult and time consuming to cross-identify these objects. State-of-the-art systems such as SkyServer.sdss.org cross-match 50m objects(GALEX) to 150m(SDSS) in 1 hour when parallelized across multiple CPUs. This problem is extremely computationally intensive, if we consider 2 catalogs, A and B, of size n and m, respectively, we need to make n*m comparisons. Which in this small case would be 7.5 Billion comparisons. We describe a method of doing a cross-match on 2 catalogs 3x larger in under 4 minutes. That yields an over **40x speedup**!

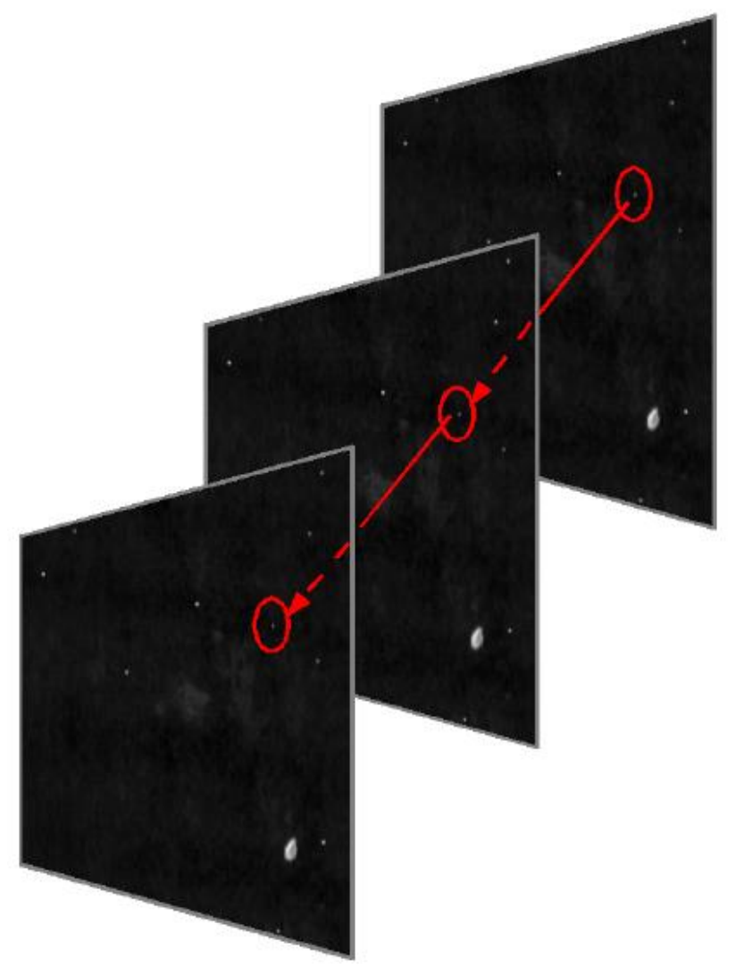


Figure 2: Example of cross-identifying the same object through several observations of the same survey

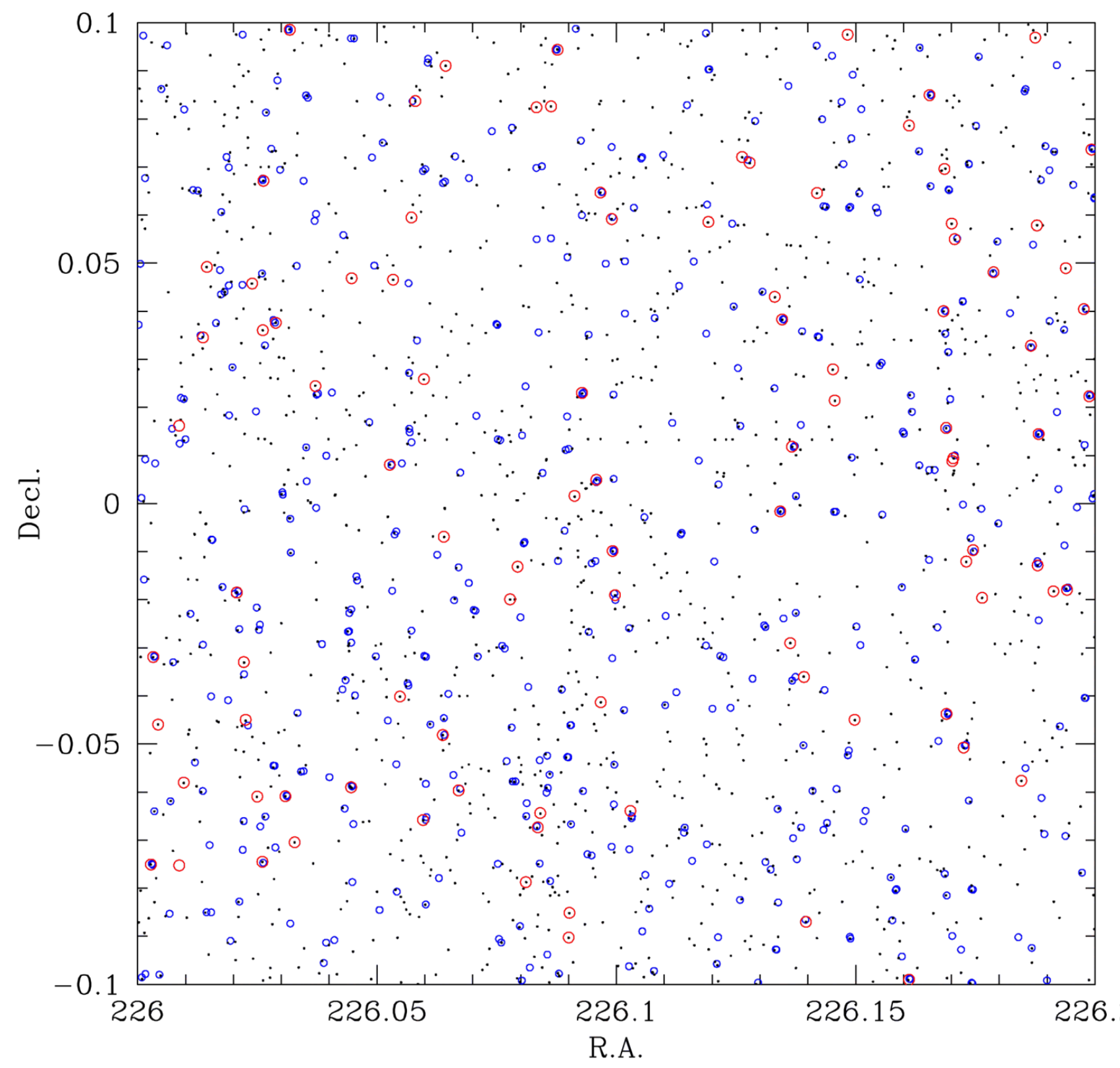


Figure 3: Excerpt of an overlapping region of the SDSS, GALEX and 2MASS surveys. Sources are color coded black, blue and red, respectively. The radii of each circle correspond to the astronomic uncertainties of σ=0.1, 0.5 and 0.8 arc seconds in the 3 surveys. Here we plot their 5σ contours for better visibility.

Divide and Conquer

The first challenge in implementing cross-matching is to adequately chunk the sky into pieces so that it parallelizes well on GPUs. As the end goal is to parallelize across not just one, but multiple GPUs, the chunking has to be effective at both the global and local scale of our processing pipeline.

The naïve approach to this problem would be comparing every datapoint of catalog A to every datapoint in catalog B. Lets remember that we are looking to identify common object across the catalogs. It is very unlikely that a common object is in completely different regions of the 2 catalogs.

Gray et al. suggest in "There Goes the Neighborhood: Relational Algebra for Spatial Data Search"[1] and "The Zones Algorithm for Finding Points-Near-a-Point or Cross-Matching Spatial Datasets"[2] to map the catalogs into horizontal zones of a height *zoneHeight*, where *zoneHeight* is measured in archseconds. (see Figure 5)

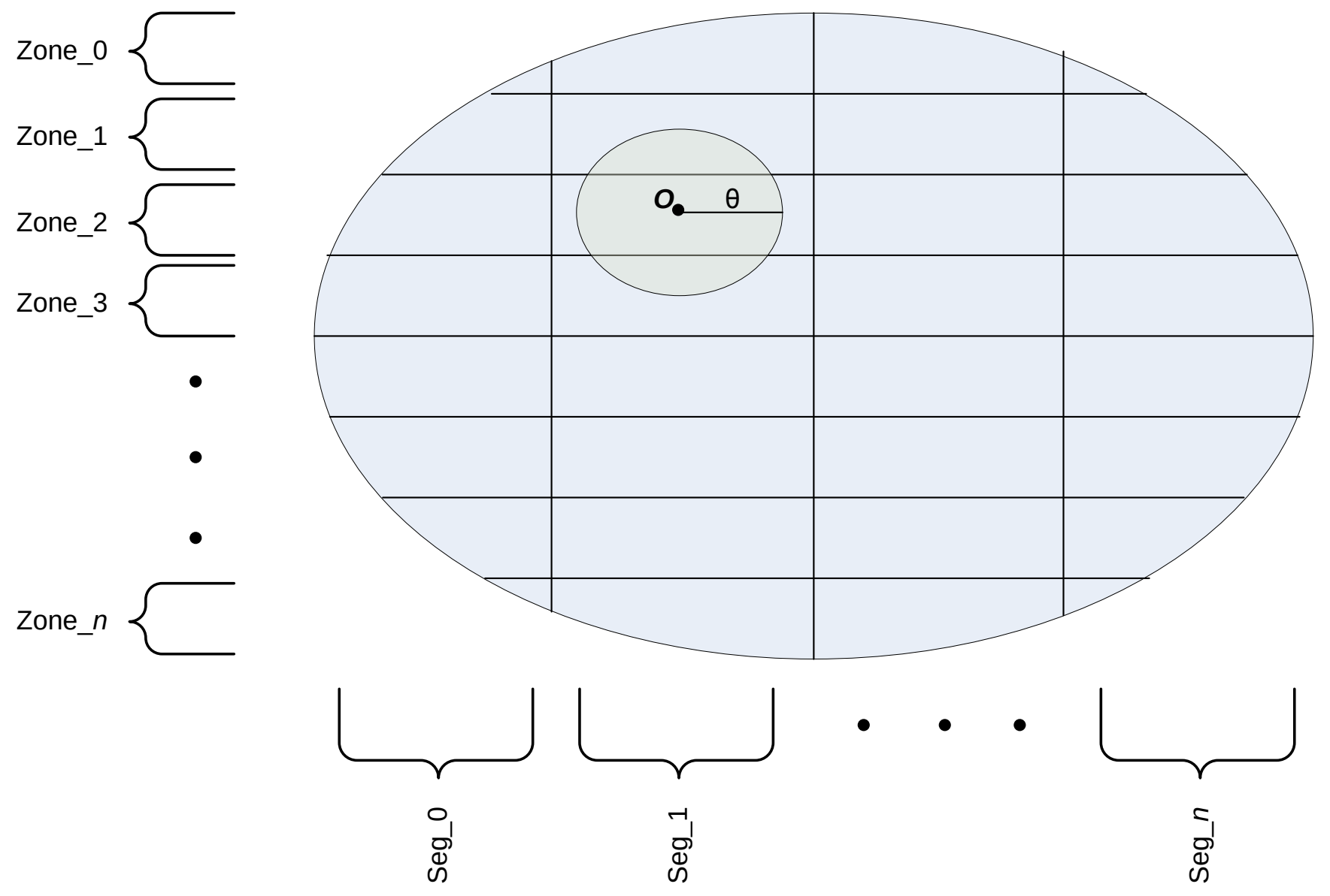


Figure 5: a Catalog divided up into Segments and Zones. Also visible is our object O and its search radius θ.

If we aim to find all matches within a radius θ of object *O*, we find the min. and max. zone enclosing our search radius in the range of declination:

minZone = [(dec - θ) / *zoneHeight*]
maxZone = [(dec + θ) / *zoneHeight*]

In Figure 5 minZone and maxZone would be Zone_1 and Zone_3. We also compute the search radius bounds in the range of RA(right ascension). Given the min and max Zones as well as the RA range, we have successfully narrowed our search field.

Parallel on Multiple GPUs

From here on we move to the parallel implementation. We start by dividing the catalog into multiple segments so that 2 segments can fit on 1 GPU at a time. We then utilize the Thrust library to sort each segment by declination(see Figure 6) to allow us to break each segment into zones.

Next we create Jobs. Each Job is a unique match between one segment of catalog A and one segment of catalog B, such that all combinations are covered. Ex: SegA_0 x SegB_0, SegA_0 x SegB_1 and so on.

These Jobs are passed off to the Job manager which stores them in a Job Queue. The Job Manager is the entity which provides jobs to each processing thread. When a processing thread finishes a Job it requests a new one. To optimize memory transfers, between the host and the device, the Job Manager prefers to give out Jobs based on which segments are left in memory from the previous Job, therefore cutting down duplicate transfers.

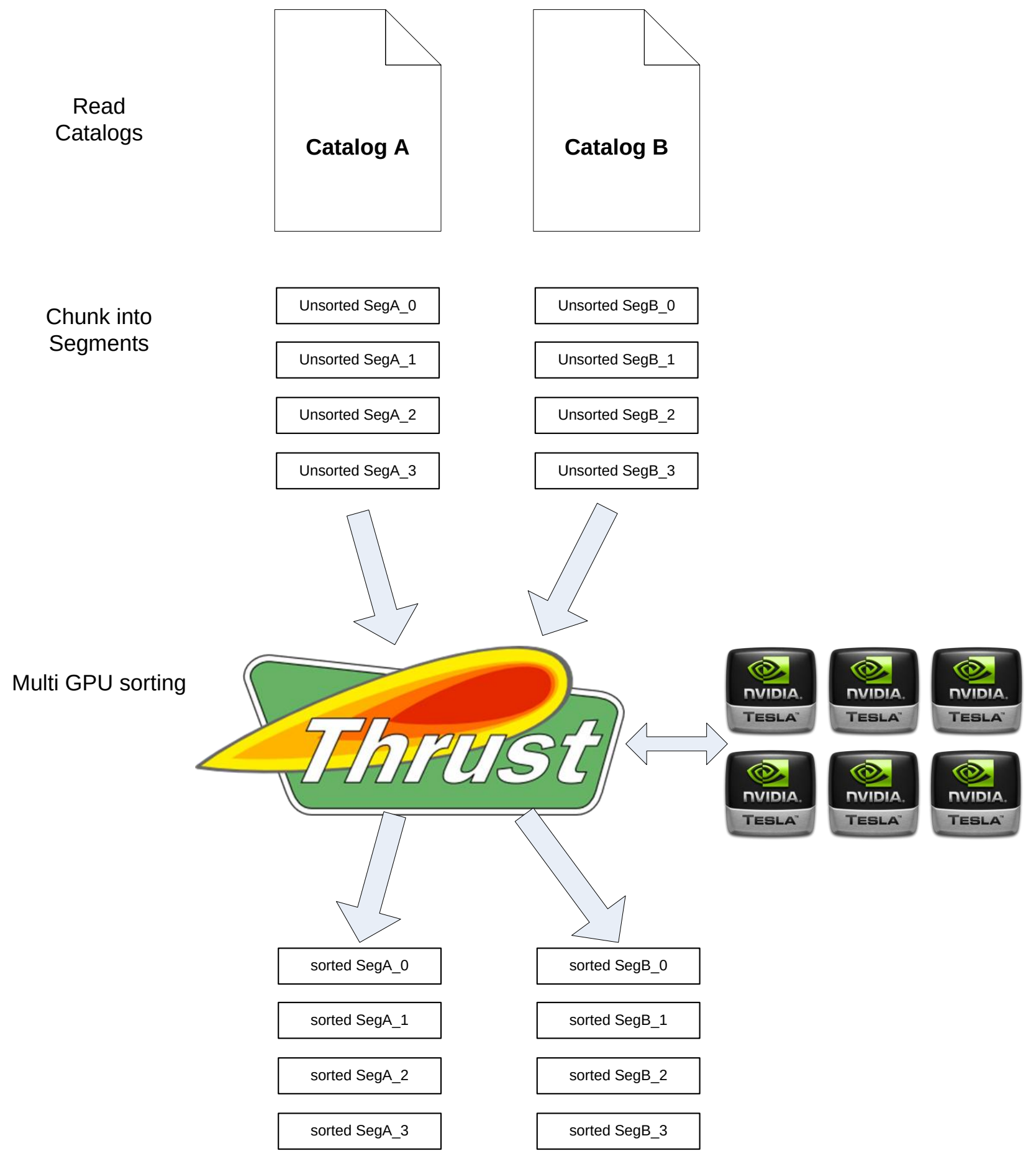


Figure 6: Reading, Splitting and Sorting of Catalogs into segments.

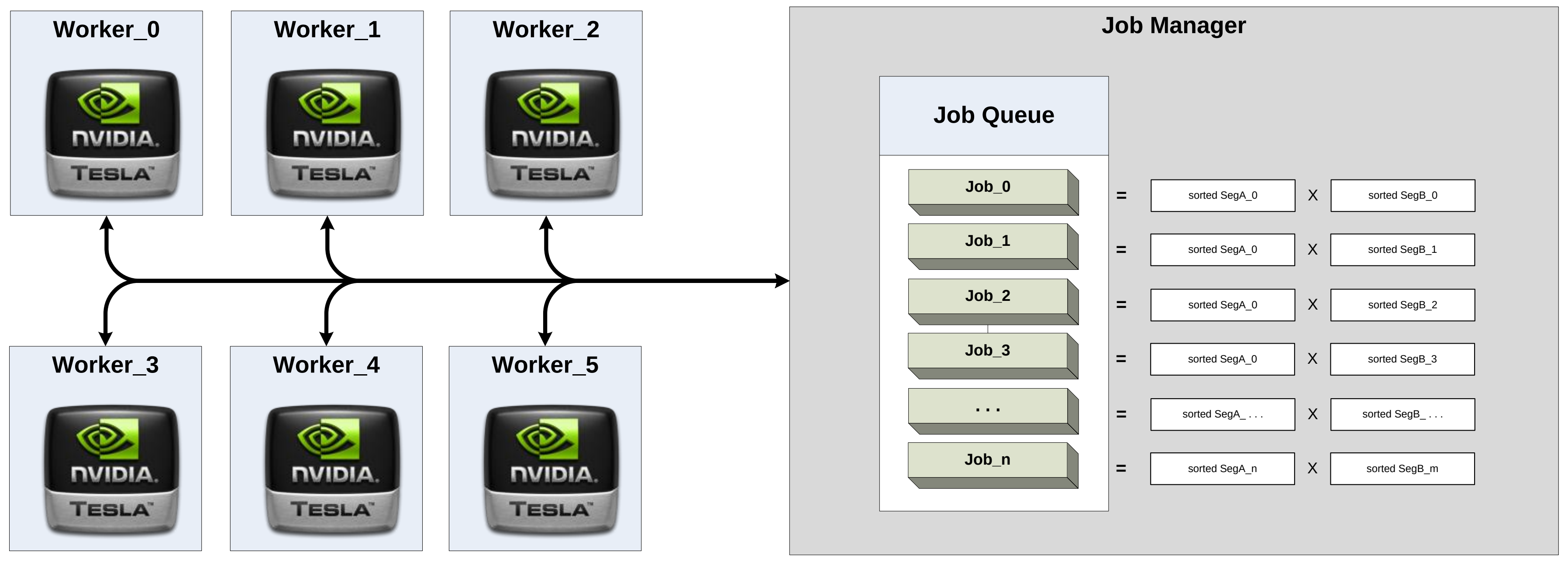


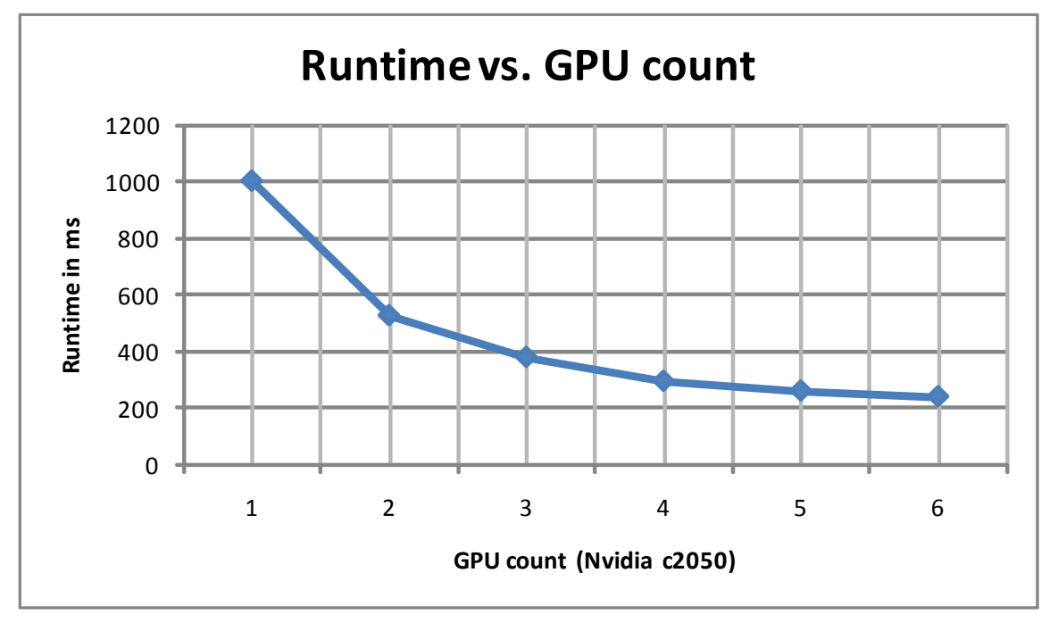
Figure 7: Job Manager holding the Job Queue and communicating with the worker threads

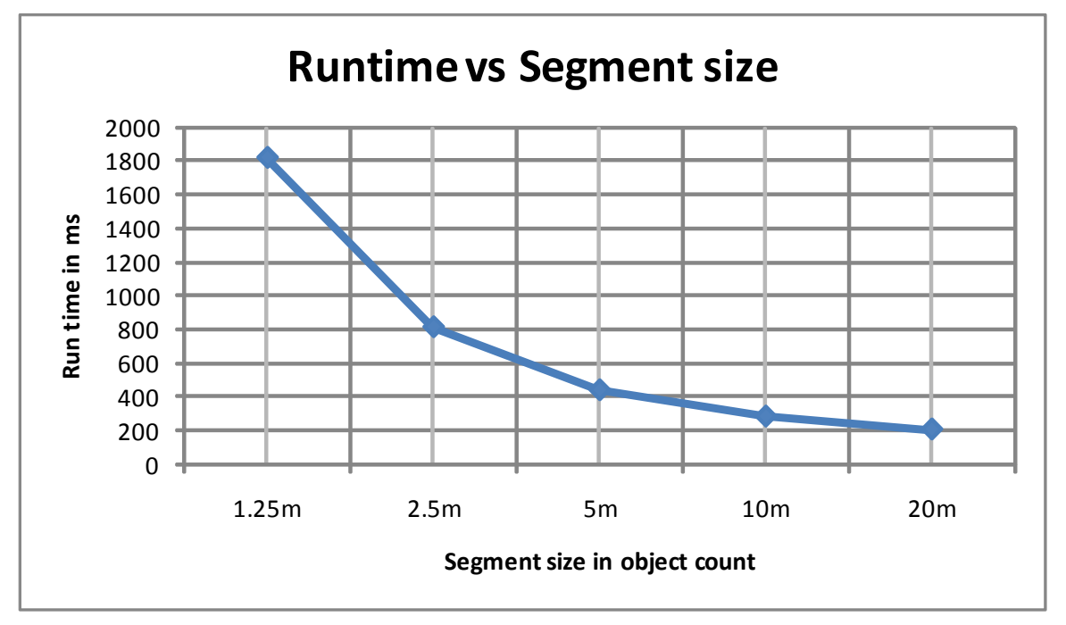
Kernel Optimization

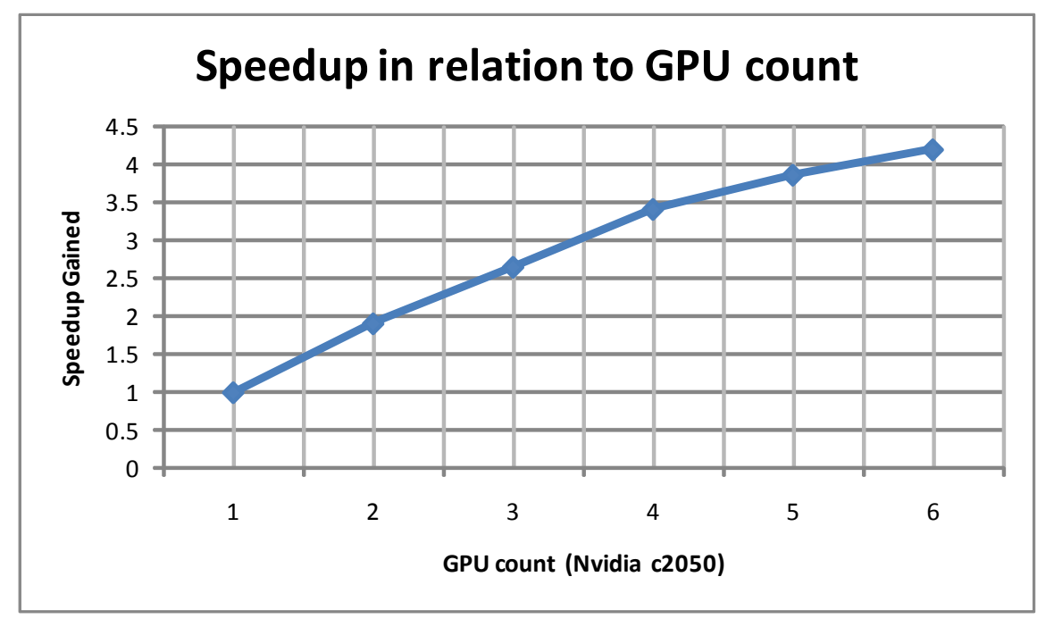
For each zone in Segment A, the worker thread launches as many kernels as there are zones within range *θ* in Segment B. Every thread of each of these kernel's calculates the vector distance between a unique pair of object, one object from each segment. If the computed distance is below a certain threshold, the result is marked "found".

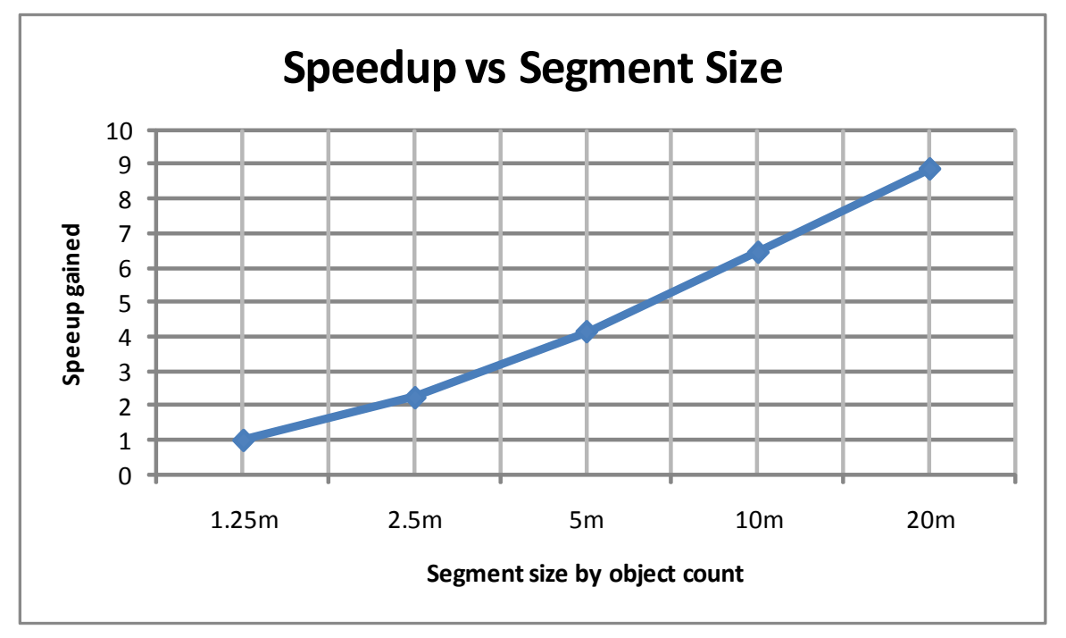
Many of the calculated distances will be greater than the threshold, resulting in a sparsely populated results array. We utilize this fact to minimize to data we need to return. In our current implementation we compact the "found" results after every kernel run into a global array using a parallel prefix scan as described in "Parallel Prefix Sum (Scan) with CUDA" by Mark Harris[3].

Results









Our results show an incredible 45x speedup over the previous best implementation based on multi CPU performance on SQL Server. Our speedups are slightly sub-linear, but that can be expected as the overhead of controlling worker threads grows, speedups decrease.

The larger the segment size we can keep in memory the better we perform on a same size problem. This is due to the overhead of launching sorters and passing jobs back and forth. Theoretically Nvida c2050s support 2 segments of 50 million objects, but we ran into glitches with our kernels and the overhead required by the Thrust library for sorting and copying segments. This is a work-in-progress.

In the future we would like to attempt adding dynamic segment size scaling based on the memory of available GPUs. We also aim to have a public release of version 1.0 in the near future. Please email us if you are interested.

References

[1] "There Goes the Neighborhood: Relational Algebra for Spatial Data Search", A.S. Szalay, G. Fekete, W. O'Mullane, M. A. Nieto-Santisteban, A.R. Thakar, G. Heber, A. H. Rots, MSR-TR-2004-32, April 2004.

[2] "The Zones Algorithm for Finding Points-Near-a-Point or Corss-Matching Spatial Datasets", J Gray, M. A. Nieto-Santisteban, A.S. Szalay, MSR-TR-2006-52, April 2006

[3] "Parallel Prefix Sum (Scan) with CUDA", M Harris, April 2007

Contact us:

Matthias A Lee
MatthiasLee@jhu.edu

Department of Computer Science
The Johns Hopkins University
3400 N Charles Street
Baltimore, MD 21238

Tamás Budavári
Budavari@jhu.edu

Department of Astronomy and Physics
The Johns Hopkins University
3400 N Charles Street
Baltimore, MD 21238