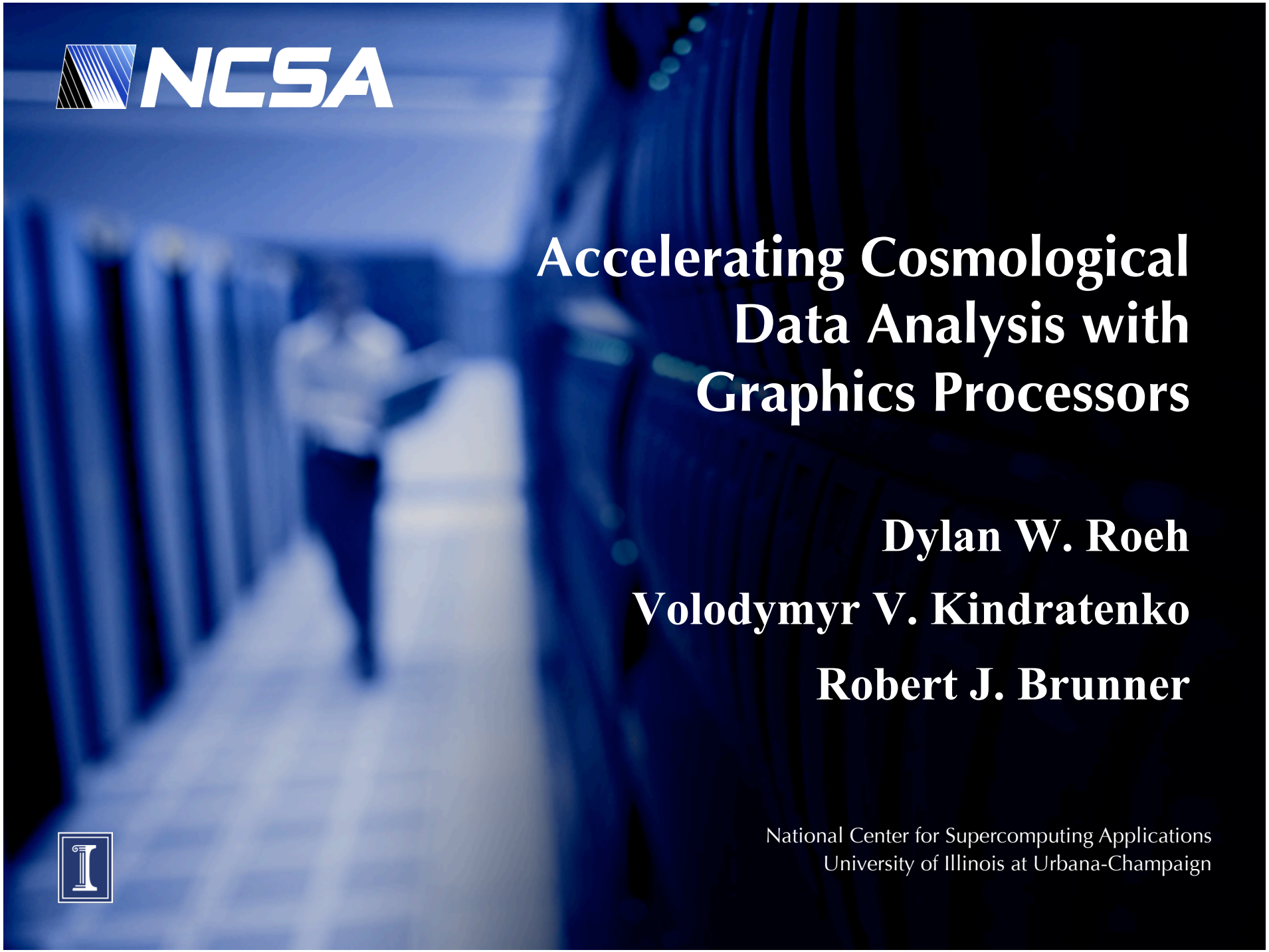


Accelerating Cosmological Data Analysis with Graphics Processors

Dylan W. Roeh

Volodymyr V. Kindratenko

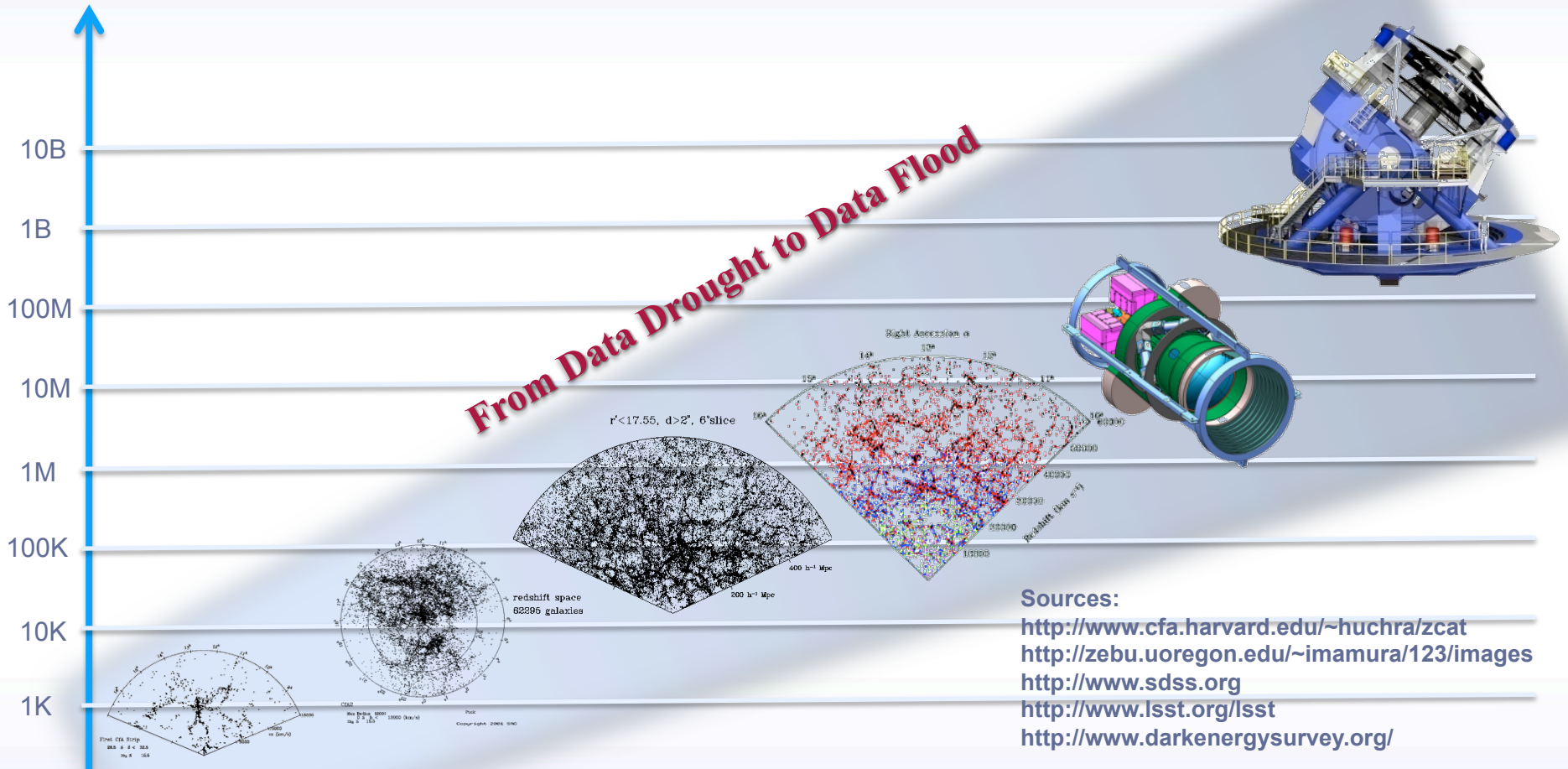
Robert J. Brunner

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

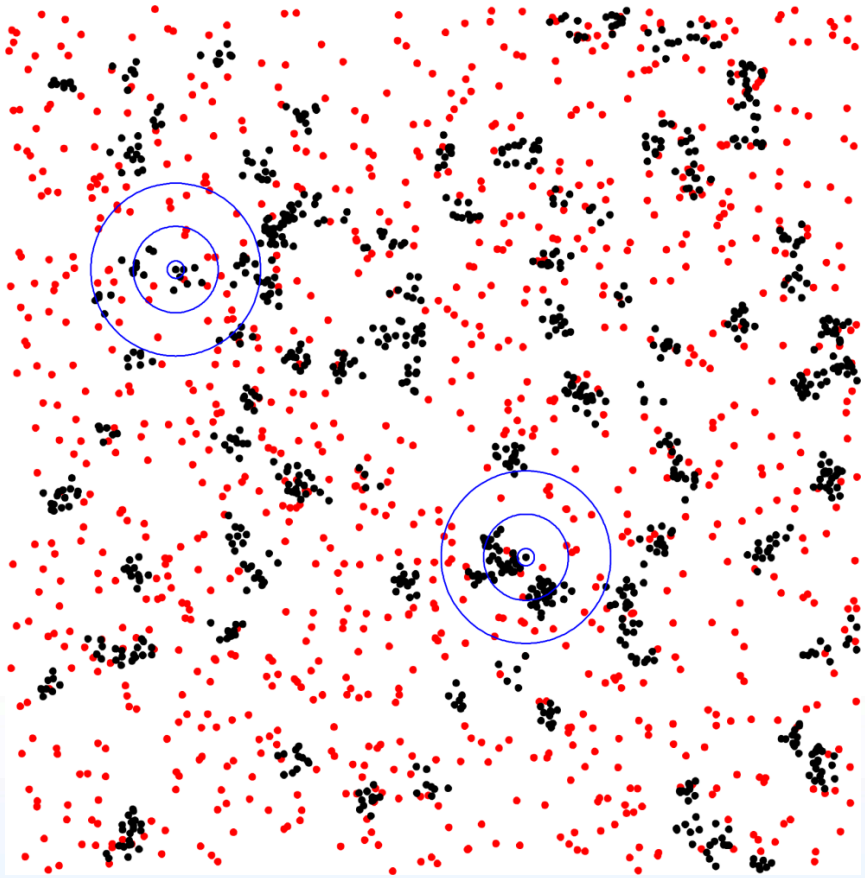
Presentation Outline

- **Motivation**
  - Digital sky surveys
- **Angular Correlation function**
  - Concept
  - Algorithm
- **GPU implementation**
  - Suitability of GPUs
  - Bin computation kernel
  - Histogram kernel
- **Results**
  - Performance on a single node
  - Multi-node scaling
  - Comparison with FPGAs
- **Conclusions**

Digit{ized | al} Sky Surveys



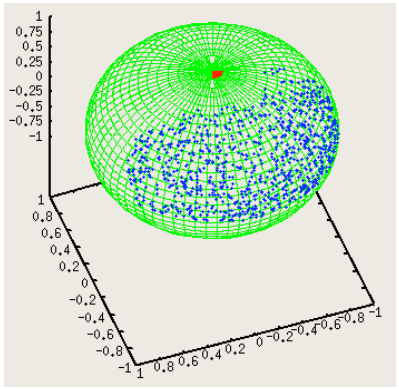
Example Analysis: Angular Correlation



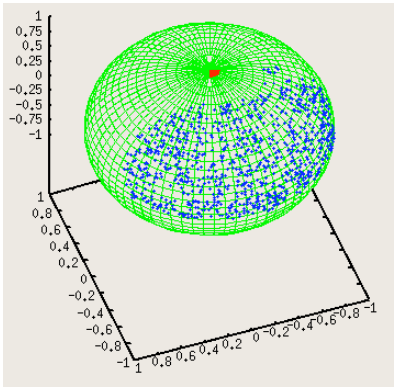
- **Two-point angular correlation function (TPACF), ω(θ), is the frequency distribution of angular separations θ between celestial objects in the interval (θ, θ + δθ)**
  - θ is the angular distance between two points
- **Red (random data) are, on average, randomly distributed, black (observed data) are clustered**
  - random points: ω(θ)=0
  - observed points: ω(θ)>0
- **TPACF can vary as a function of angular distance, θ (blue circles)**
  - random: ω(θ)=0 on all scales
  - observed: ω(θ) is larger on smaller scales

Two Point Angular Correlation Function

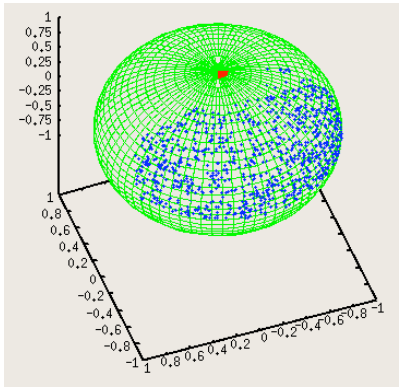
observed data



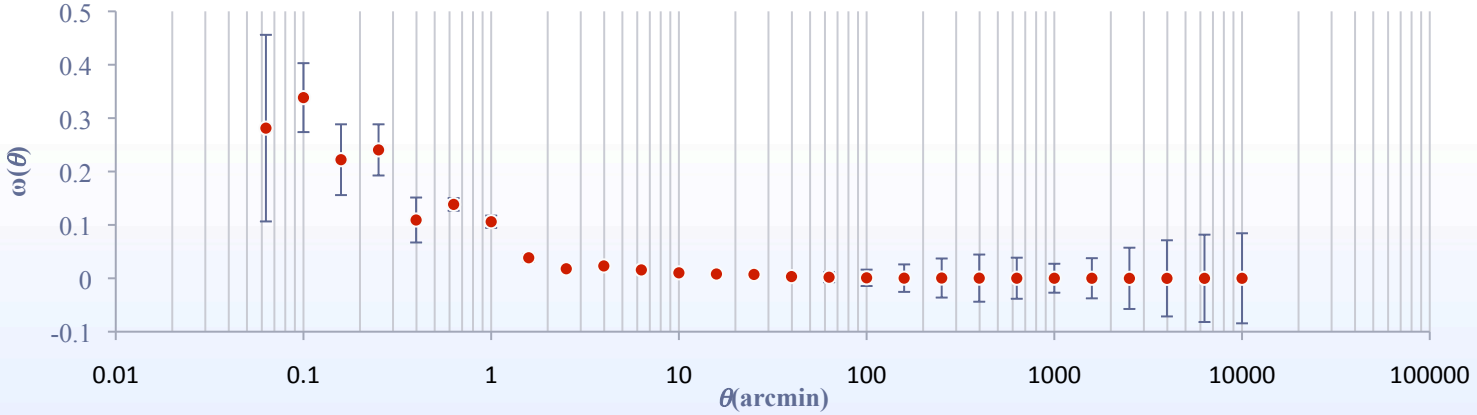
random dataset 1



random dataset n_R



TPACF



TPACF Algorithm

- **Modified Landy & Szalay estimator**

$$\omega(\theta) = \frac{n_R \cdot DD(\theta) - 2\sum_{i=0}^{n_R-1} DR_i(\theta)}{\sum_{i=0}^{n_R-1} RR_i(\theta)} + 1$$

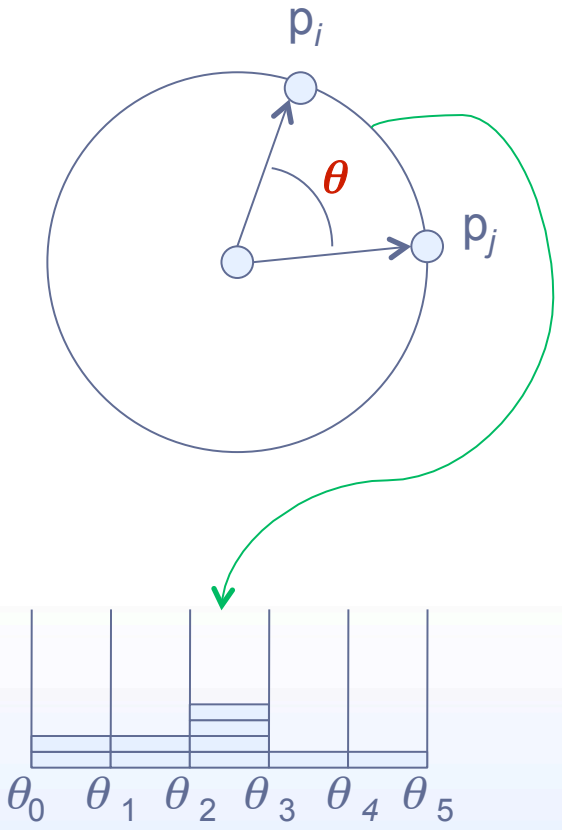
- **Angular distance**

$$\theta = \arccos(p \cdot q) = \arccos(x_p x_q + y_p y_q + z_p z_q)$$

- **Bin edges**

$$binedge_j = \cos(10^{\log_{10}\theta_{\min} + j/k}),\ j = 0,\ldots,M$$

- **Computing DD/DR/ RR counts**



TPACF Algorithm

- **Error estimation via jackknife re-sampling**
  - This technique first divides the area of interest into multiple subsamples, after which, single subsamples are systematically removed, one-at-a-time, and the correlation function is estimated for the remaining subsamples
  - In practice, this is implemented by labeling each observed object with the sample number from which it is removed and updating only those histograms of angular separation, $DD(\theta)$ and $DR(\theta)$, that do not belong to the given sample

TPACF Algorithm Implementation

```
// pre-compute bin boundaries, binb

loadObservedData(data);
computeDD(data, npd, data, npd, 1,binb, nbins, njks, DD);

for (i = 0; i < random_count; i++) // loop through random data files
{
      loadRandomData(random[i]);
      computeRR(random[i], npr[i], random[i], npr[i], 1, binb, nbins, njks, RRS);
      computeDR(data, npd, random[i], npr[i], 0, binb, nbins, njks, DRS);
}

// compute w
for (k = 0; k < nbins; k++)  {
      ω[k] = (random_count * 2*DD[k] - DRS[k]) / RRS[k] + 1.0;
}
```

TPACF Algorithm Implementation

```
void compute{DD|DR|RR}(struct cartesian *data1, int n1, struct cartesian *data2, int n2,
                       int doSelf, int nbins, double *binb, int njk, long long **data_bins)
{
  if (doSelf) { n2 = n1; data2 = data1; }            // setup pointers for Self-compute

  for (i = 0; i < ((doSelf) ? n1-1 : n1); i++) {     // loop over points in the first set
    double xi = data1[i].x;                          // get point from first dataset
    double yi = data1[i].y;
    double zi = data1[i].z;
    int jk = data1[i].jk;

    for (j = ((doSelf) ? i+1 : 0); j < n2; j++) {    // loop in second dataset
      double dot = xi * data2[j].x + yi * data2[j].y + zi * data2[j].z;   // dot product

      int indx, k = nbins;                           // find bin it belongs to
      if (dot >= binb[0]) indx = 0;                  // eliminate those outside the range
      else {  while (dot > binb[k]) k--; indx = k+1; }   // sequential search

      for (l = 0; l < njk; l++)
        if (l != jk) data_bins[l][indx] += 1;        // update all but jk bins
    }
  }
}
```

Suitability of GPUs

- **The computation of histogram bin counts is highly parallel with substantial data reuse opportunities**
  - Each point is used N_D times during invocation of the kernel
  - Each pair of points can be treated independently, allowing $O(N_D^2)$ parallel calculations
  - Assuming that a group N_t~100 points can be pre-loaded and used N_t times ($N_t << N_D$) in the dot product calculation, $8(N_t+3)$ load/store operations are required per $5N_t$ floating-point operations, resulting in $8(N_t+3)/(5N_t) \approx 1.6$ bytes per flop (assuming double-precision)
  - This rate is comparable to the peak rate of ~1.8 bytes per flop on NVIDIA GeForce GTX280GPU (assuming double-precision)

GPU Implementation Overview

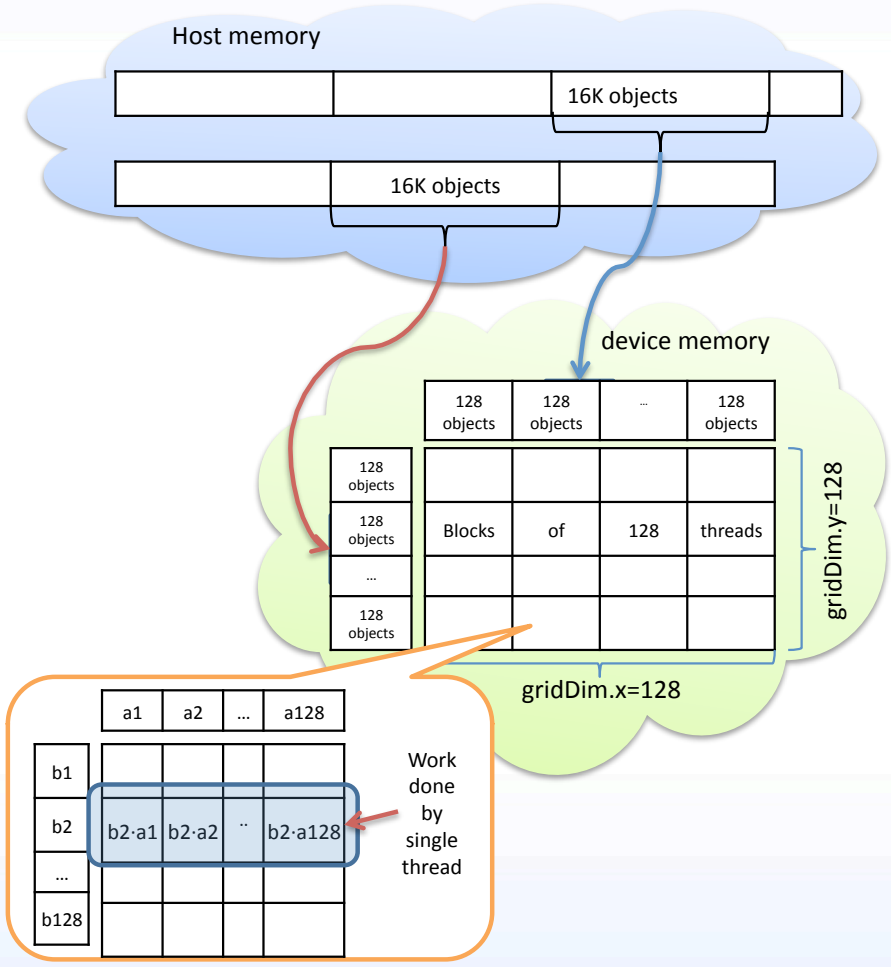
- **The algorithm is broken up into two major kernels**
  - The first kernel takes two sets of 3D vectors, computes the dot product of every pair of vectors, and writes histogram bin assignments to memory
  - The second kernel reads bin assignments from memory and constructs a histogram
    - This actually uses a third kernel; a helper kernel to compile sub-histograms in device memory
- **Histogram kernel is only called on elements with the same jackknife index. By doing this we can reconstruct the full histogram and all jackknife histograms without many redundant calls to the histogram kernel.**

Difficulties

- **Computing a bin assignment from the result of a dot product**
  - Direct bin index computation involves computing logarithm and arccosine, which is slow
  - Binary and linear search based bin index computation suffer form branch divergence
- **Histogram algorithm**
  - Avoiding race conditions in global memory requires constructing per-block histograms
  - Avoiding race conditions within shared memory is only possible by creating per-thread histograms in shared memory
    - Must be careful not to exceed the maximum shared memory usage
    - Some trickery is required to avoid bank conflicts

Bin Computation Kernel

- This kernel takes in two lists of vectors and computes the dot product of every pair of vectors, and then each dot product's bin assignment. The bin assignments are then written to a grid in device memory
  - It is assumed that the lists both have 16,384 elements, but this is not necessary for the algorithm in general
- Each block has 128 threads, and each thread computes and outputs 128 bin assignments
- Number of bins is assumed to be small enough that 4 bin assignments can be packed in an integer



Bin Computation Kernel

- **A linear search proves to be the best method for computing a bin assignment given a dot product**
- **Note that, when computing DD or RR*i*, it suffices to compute bin assignments for fewer than half of the pairs**
  - Doing this introduces branch divergence only in blocks for which blockIdx.x = blockIdx.y; the rest must either compute all bin assignments or no bin assignments
  - This also removes troubling elements on the main diagonal, which tend to be incorrectly binned in single precision implementations
  - Note that we must reserve a histogram bin for “ignored” elements

Histogram Kernel

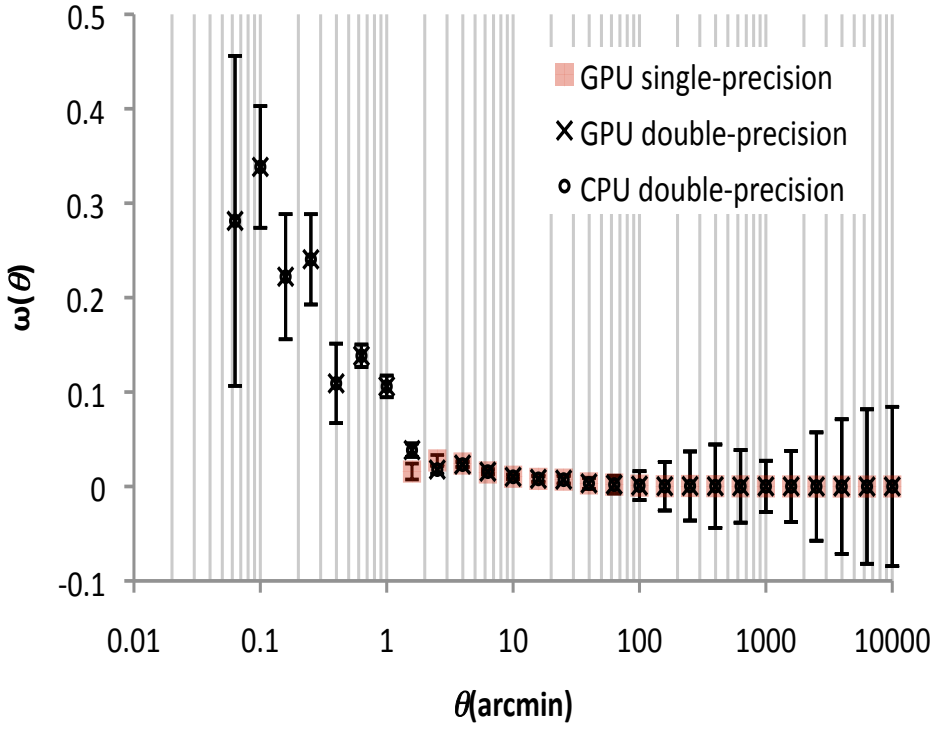
- **The histogram kernel (based on NVIDIA's whitepaper) functions by first computing per-thread sub-histograms, then compiling those into per-block sub-histograms and writing them to global memory**
- **Following this, a small helper kernel compiles the per-block sub-histograms into a smaller number of sub-histograms if necessary. Compiling the small number of remaining sub-histograms into a full histogram is left to the CPU**
  - The helper kernel can be eliminated on Compute Capability 1.1 or greater GPUs with the use of atomic memory operations, but doing so results in some loss of performance
- **The per-thread sub-histograms must be stored in shared memory. Given that, we want to have a reasonable number of threads without putting overly harsh restrictions on the number of bins or maximum capacity of a given bin**
  - Using one byte to represent a histogram bin allows each thread to histogram up to 255 elements
  - Doing this we can achieve 64 bins with 192 threads without over-running shared memory
  - 64 is plenty for this application. 128 bins could be achieved with the same algorithm, but would require a reduction to 64 threads

Jackknife Resampling

- **Part of the goal of this implementation was to include jackknife re-sampling, in which we compute not only the full histogram for $\omega(\theta)$, but also a number of sub-histograms (jackknives) which are to be used in error bounds**
  - Each element is removed from precisely one jackknife
- **The obvious implementation is to add each element to the full histogram as well as every jackknife except that which it is removed from**
  - Unfortunately, this requires either far too many bins (e.g., 330 if 30 bins and 10 jackknives are used, as in the CPU version) or many redundant calls to the histogram kernel
- **An efficient solution is to use "inverse" jackknives. The *i*th inverse jackknife is the histogram of elements which are removed from the *i*th jackknife**
  - The full histogram and every jackknife can be easily reconstructed
  - Every element goes through the histogram kernel precisely once
  - May introduce some calls to the histogram kernel which are not necessary without jackknife re-sampling, but the histogram kernel does not have much overhead, and the number of extra calls is relatively small

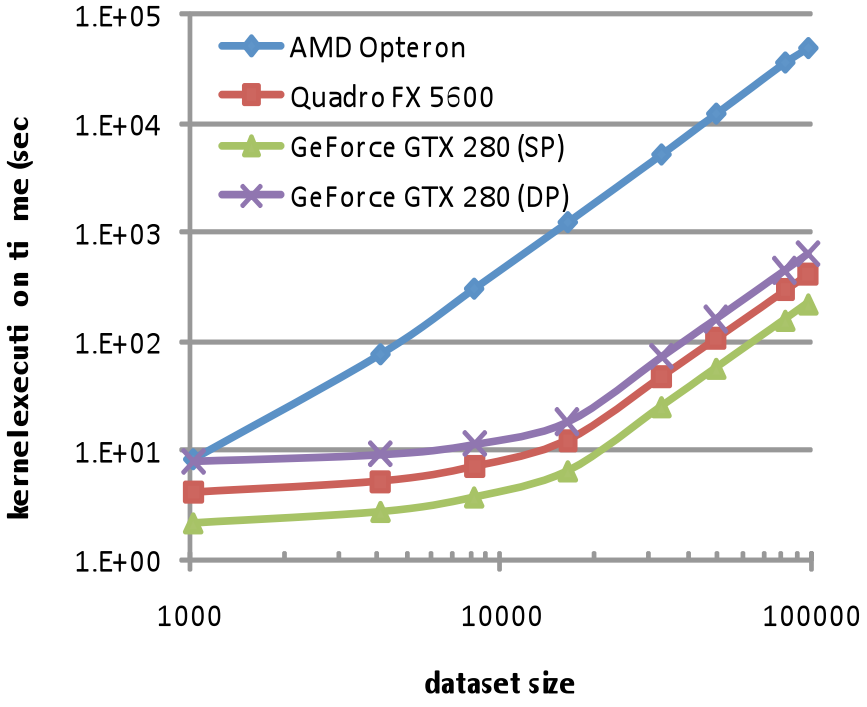
Singe/double-precision Issue

- The double-precision results for the GPU and CPU implementations coincide
- The single-precision GPU results are off for angular separations below approximately 4 arcminutes
- The numerical precision of single-precision floating-point arithmetic is not sufficient to perform the distance calculations for angular separations below approximately 1 arcminute

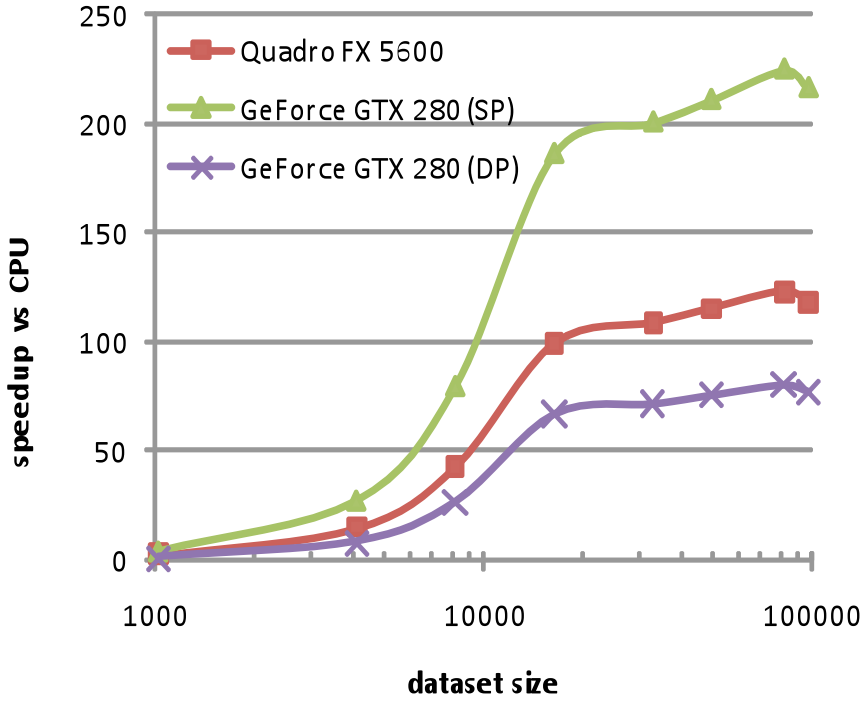


Performance on a Single GPU

- Execution time

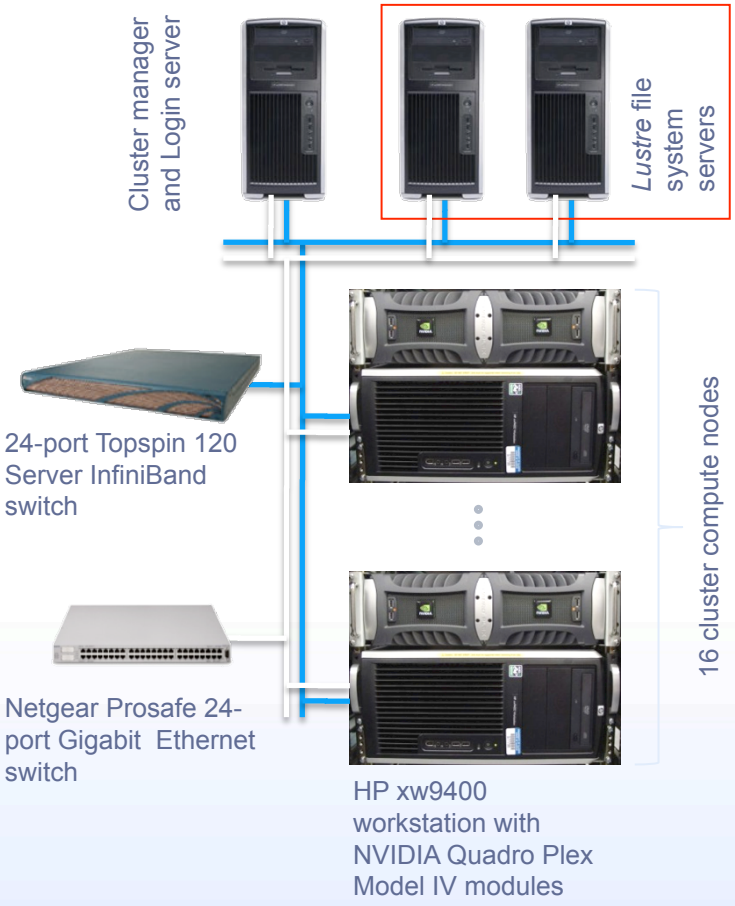


- Speedup

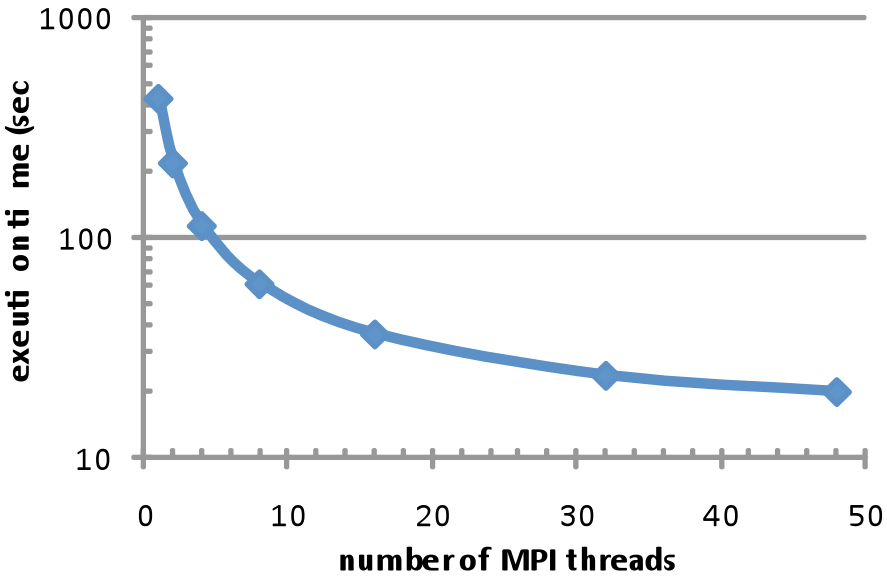


Multi-GPU MPI Implementation

- GPU cluster



- Performance scaling



Comparing with FPGAs

Measured features/ parameters	SRC-6 host *2.8 GHz Xeon*	SRC-6 dual-MAP	SGI Altix host *1.4 GHz Itanium 2*	RC100 blade
# CPUs	2		2	
# FPGAs		4		2
# of compute engines	1	17	2	4
DD time (s)	219.5	3	226.6	49.7
DR+RR time (s)	84,354.3	880.3	47,598.6	4,975.3
Load/convert (s)	20.3	20.7	28.4	27.5
Total (s)	84,594.1	904	47,853.6	5,052.5
Overall Speedup	1.0×	93.5× (1) 52.9×	1.0×	9.5× (2)

(1) **V. Kindratenko**, R. Brunner, A. Myers, *Dynamic load-balancing on multi-FPGA systems: a case study*, In Proc. 3rd Annual Reconfigurable Systems Summer Institute - RSSI'07, 2007.
(2) **V. Kindratenko**, R. Brunner, A. Myers, *Mitrion-C Application Development on SGI Altix 350/RC100*, In Proc. IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'07, 2007.

Conclusions

- **Significant performance improvements achieved on GeForce GTX 280 GPU as compared to a 2.4 GHz AMD Opteron-based system**
  - 225× speedup for single-precision implementation
  - 80× speedup for double-precision implementation
  - Work that requires days on a PC, or hours on a small cluster, now can be done in minutes on a GPU!
- **Code scales, to a limit, on a multi-GPU system using MPI**
- **Programming effort equals to one semester of a good graduate student working part-time**

Acknowledgements

- **This work has been supported by the NSF STCI (OCI 08-10563) and NASA AISR (NNG06GH15G) programs. The GPU cluster used in this work is funded by the NVIDIA CUDA Center of Excellence at the University of Illinois and the NSF CRI (CNS 05-51665) grant under the CRI program. The cluster is managed by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) as part of the Institute for Advanced Computing Applications and Technologies (IACAT).**