NCSA GPU programming tutorial day 4

Dylan Roeh

droeh2@illinois.edu

Introduction

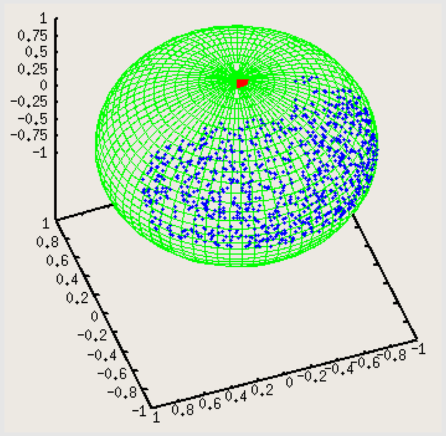
- Application: Two Point Angular Correlation Function
- The goal is, given a large set *D* of unit vectors, to compute a histogram of *ω(θ)* for various values of *θ*
- To do so, we first compute a large number of dot products and create a number of histograms: *DD, DR$_i$, RR$_i$*. Each *R$_i$* represents a large set of random unit vectors. *DD* is the histogram of all dot products of pairs of elements in *D*, and similarly with the others

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

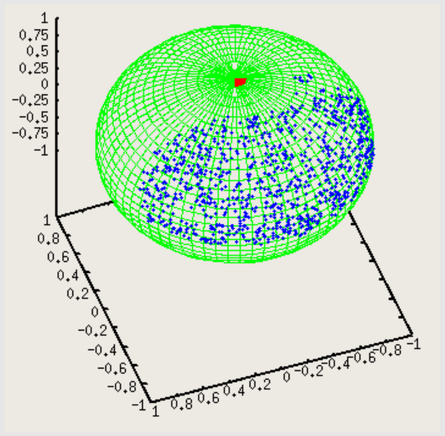
- Then perform some operations on these histograms to compute a histogram of *ω(θ)*
- Jackknife resampling also required

Two Point Angular Correlation Function

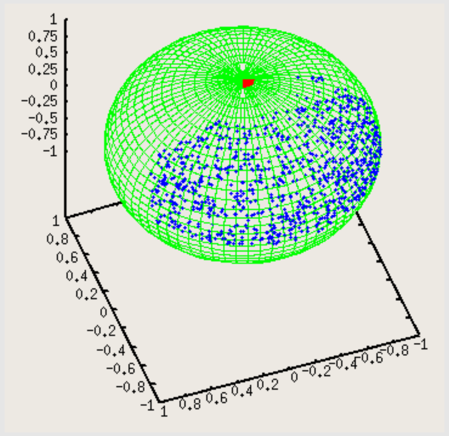
observed data

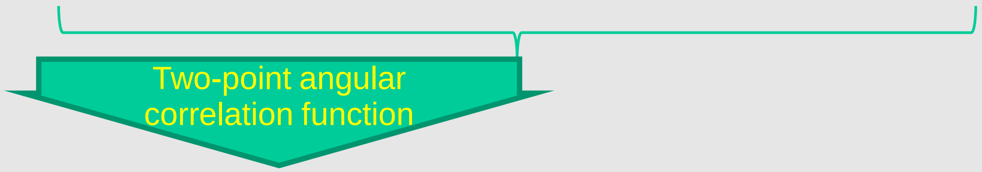


random dataset #1

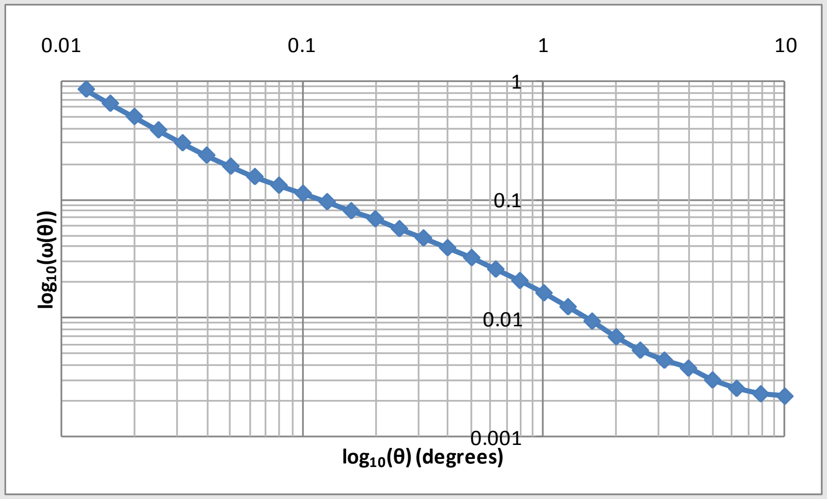


random dataset #100





…

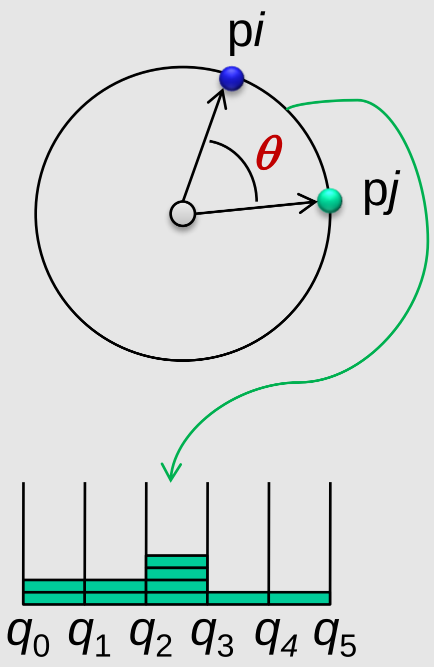


Overall Code Organization

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

load_file(data); // load data from disk  (~100K-1M points on a sphere)
doCompute (data, npd, data, npd, 1, DD, binb, nbins); // compute DD

for (i = 0; i < random_count; i++) // loop through random data files
{
    load_file(random[i]); // load data from disk (~100K-1M points on a sphere)
    doCompute (random[i], npr[i], random[i], npr[i], 1, RRS, binb, nbins); // compute RR
    doCompute (data, npd, random[i], npr[i], 0, DRS, binb, nbins); // compute DR
}

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

doCompute kernel

```
void doCompute(struct cartesian *data1, int n1, struct cartesian *data2, int n2, int doSelf, long long
     **data_bins, int nbins, double *binb, int njk) {

  if (doSelf) { n2 = n1; data2 = data1; }  // setup pointers for doSelf compute mode

  for (i = 0; i < ((doSelf) ? n1-1 : n1); i++) {  // loop over points in the first dataset

    double xi = data1[i].x; 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 over points in the second dataset

      double dot = xi * data2[j].x + yi * data2[j].y + zi * data2[j].z;    // compute dot product

      int  indx, k = nbins;

      if (dot >= binb[0]) indx = 0; // data_bins[0] += 1;   // find bin it belongs to

      else { while (dot > binb[k]) k--;  indx = k+1; // data_bins[k+1] += 1; }

      for (l = 0; l < njk; l++) // update all samples

        if (l != jk) data_bins[l][indx] += 1;

    }

  }

}
```



Applicability to GPUs

- The major parallel aspects of the algorithm are the computation of the histogram bin assignments and the actual computation of the histograms
- The former is fairly straightforward, as it essentially requires taking a very large number of independent dot products, and assigning bins based solely on the result of said dot products
- The latter can be implemented in a straightforward fashion, but achieving good performance requires more careful consideration
- Other parallelism exists, such as in looping over all values of *i*. This is best taken advantage of in a multi-GPU system

Implementation Overview

- The algorithm is broken up into two major kernels
  - The first takes two sets of 3-vectors, computes the dot product of every pair of vectors, and writes histogram bin assignments to memory
  - The second 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

- The first difficulty which arises is the problem of computing a bin assignment from the result of a dot product. The CPU version implements both linear and binary searches, as well as a direct bin computation formula
  - Direct computation seems as though it would be best; but invocation of logarithm and arccosine render it slower than the searches. Between binary and linear, linear turns out to be best, likely due to “top” bins containing a vast majority of elements. The bias reduces the likely number of control paths as well as the number of *if*s to be executed within a given control path.

Difficulties (cont.)

- The second difficulty is efficiently implementing a histogram algorithm.
  - Avoiding race conditions in global memory or poor global memory bandwidth requires that we construct 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
- In this implementation 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 (cont)

- As mentioned previously, 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 an nVidia 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
  - We could eliminate the helper kernel on Compute Capability 1.1 or greater GPUs with the use of atomic memory operations, but doing so results in some loss of performance

Histogram Kernel (cont)

- 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 resampling, 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 (330 if 30 bins and 10 jackknives are used, as in the CPU version) or many redundant calls to the histogram kernel

Jackknife Resampling (cont)

- An efficient solution is to use “inverse” jackknives. The ith inverse jackknife is the histogram of elements which are removed from the ith 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 resampling, but the histogram kernel does not have much overhead, and the number of extra calls is relatively small

Multi-GPU Implementation

- As mentioned earlier, the *i*-loop can be parallelized to allow for easy multi-GPU implementation. This can be done with pthreads, but the current implementation uses MPI
- Using multiple GPUs in MPI is easy once each process has its own GPU
  - Unfortunately, assuring this can be tricky. The simplest method is to use the same number of GPUs on every machine, and ensure that process IDs on a given machine are sequential. With these assumptions, one can simply use the process ID modulo the number of GPUs per machine to assign a unique GPU to each process