T P A C F

(Two-Point Angular Correlation Function)

Scott Bai, Ali Hussain, Gene Wu, Nicolas Zea

Outline

- TPACF Defined
- Parallelism in TPACF
- Single-Precision Issues
- Implementation Version 1
- Implementation Version 2
- Performance Comparisons
- Conclusion

TPACF Defined

- ***Calculates frequency distribution of angular separations between data coordinate positions, compared to randomly distributed positions in the same space [1]***
- ***Correlation functions are used in a variety of scientific disciplines***
- ***Specific application: for Sloan Digital Sky Survey, calculates probability that a star or galaxy will be found at a given angular distance from another [1] astronomical object***

[1] 2pacf Project Application Information Sheet.pdf

TPACF Defined

- Inputs
  - One set of spherical coordinates representing astronomical bodies, with M data points
  - N randomly generated sets of spherical coordinates with M data points each

TPACF Defined

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

[1]

- DD: data points correlated with self
- DR: data points correlated with a random set
- RR: set of random points correlated with self

[1] 2pacf Project Application Information Sheet.pdf

TPACF Defined

- Steps involved:
  - Conversion of data and random points from spherical to Cartesian coordinates
  - Calculate DD, DR, and RR
    - Dot product
    - Binning of results into histogram
  - Summations of DRs and RRs for each theta value
  - Calculation of w(theta)

Parallelism in TPACF

- Conversion of data points
- All dot products of DD, DR, and RR are independent both within data sets and across data set pairs
- The creation of histograms is difficult because we need to load a bin value from shared memory, increment the value, and store the value back without losing values due to thread-scheduling race conditions

Single-Precision Issues

- The gold code relies on double precision floating point values to represent both the coordinates of the data points, the bin boundaries of the histograms, and the bin count within the histograms
- At least 41 bits of fixed-point precision are required to reproduce the results of the gold code with complete accuracy.
- Only single point precision is supported in CUDA for now

Single-Precision Issues

- Histogram bin boundaries are exponentially increasing
- With double point precision thirty distinct bins can be represented
- Lower bins cannot be differentiated in single precision
- Conversion from double to single floating point precision results in only 20 bins
- Loss of the lower ten bins (but these held only a small percentage of distances)

Implementation Version 1

- Two kernel functions
- One kernel processes the data
  – Each call correlates one DD, DR, or RR
  – One call for DD
  – N calls for DRs, one for each R; similarly for Rrs
  – MxM coordinate pairs to correlate in each call
  – Autocorrelations (DD, RRs) treated like any others
    - does twice as much work as neccessary

Implementation Version 1

- Each warp generates a histogram using synchronization technique from class slides
  - Each block coalesces its warp histograms into one output histogram
- Second kernel function is called multiple times to coalesce histograms generated by first kernel
  - Summation is by binary tree, similar to scan
  - log(x) iterations, where x is num of blocks used in first kernel function

Implementation Version 1

- Finally, results from DRs and RRs combined into their summation terms
- w(theta) calculated for each theta (each histogram bin) from DD term and DR, RR summation terms

Implementation Version 2

- Exploit Parallelism between data sets
- 1 kernel call
- 201 blocks (1 DD, 100 DR, 100 RR) with 100 random datasets
- Each DR block does (number of elements)^2 calculations
- DD and RR around half that

Implementation Version 2

- Calculations left on CPU
  - Conversion of data points from Spherical to Cartesian coordinates
  - Summing of DR and RR sets for omega function calculation
- Done due to relatively small number of calculations performed (for summing only 100 summations for N=100 random sets) versus high overhead

Implementation Version 2

- Bring BLOCK_SIZE elements from both data sets into shared memory at a time
- For each element in first data set get dot product with all elements from second set
- Get the next BLOCK_SIZE elements
- Requires NUM_ELEMENTS global loads for first dataset
- $(\text{NUM\_ELEMENTS})^2/\text{BLOCK\_SIZE}$ global loads for second

Implementation Version 2: Advantages

- Easier to program
- Less overall code (1 vs 2 kernels)
- Reduced overhead:
  - Single kernel call
  - Fewer histograms to coalesce

Implementation Version 2: Disadvantages

- Handling of larger values slightly more difficult
- Higher register usage (approaching SM limit)
- Only one block can be resident on an SM at a time
- Parallelism does not scale

Histogram Correctness

- Algorithm from lecture slides:
  - Increment bin and write to memory with a thread tag in upper bits
  - Read memory and make sure value is correct and thread tag matches
  - Repeat until match
- Compiler optimized away loading from shared memory in loop condition check
- Solution: declare warp histogram as “volatile __shared__”

Histogram Performance

- Performance depends on number of histogram binning conflicts (needing to rewrite bin info)
- More histograms per warp = less binning conflicts
- For 10 datasets and 2048 points:
  - 280ms for 1 histogram per warp
  - 200ms for 2 histograms per warp
  - 150ms for 4 histograms per warp
  - 115ms for 8 histograms per warp*
  - 100ms for ideal zero conflict scenario
- Effects of conflicts mitigated

* Kernel failed to run in feature rich version, most likely due to too high register usage

Final Results

- Implementation 2 easily achieves speedup, 1 only for large datasets
  - But only when bin counts fit within 30 bits of precision (full data set results in overflow)
  - DD/DR/RR values are not exact though
  - Particularly in lower bins, difference from gold code varies, but stays within 10%
  - Upper bins exact or <1% difference
  - Due to GPU floating point implementation differences (emulation is exact)

Final Results

- For case of 2048 points per data point set, and 10 random sets
  - Gold: 2 seconds
  - Implementation 2: 0.115 second
  - 17x speedup
- Full data set values not found due to running time increasing exponentially (> 7 seconds leads to crashes)
  - But speedup increases the more data sets available (more blocks available)
- Register usage causing kernel failures and histogram binning method were limiting factors
  - V2 performance speedup were still near ideal

Future Optimization Ideas

- Break the higher order bins into multiple bins
  - Less binning conflicts
- Use texture memory to implement a table lookup on higher order bins (replaces binary search and check against bin boundaries)
- Try an alternative histogram binning scheme that efficiently handles conflicts

TPACF Thoughts

- TPACF lends itself readily to parallel implementation
  - Large number of independent floating point calculations
  - Data can be logically partitioned (DD, DR, RR)
- Difficulties:
  - Shared memory usage and register usage were pushed to limits
  - Histogram implementation (no atomic increment)

G80 Programming Thoughts

- Kernel crashes:
  - Very difficult to debug
  - Often leaves memory untouched, so can mislead into thinking the kernel completed successfully
  - Kernel crash error messages largely unhelpful
- Need for volatile keyword with shared memory unexpected (assumed unnecessary)
- Inability to debug shared memory usage in emulation mode was an annoyance (forced to use printfs or explicitly store in registers)

Conclusion

- Algorithm successfully ported to CUDA, with 32 bit precision and resource usage being greatest limiting factors for performance and accuracy
- Future hardware versions should address this
- CUDA shown to be capable of improving correlation function calculation greatly