Scott Bai, Ali Hussain,
Nicolas Zea, Gene Wu
Final Project Report
Due: May 9th, 2007

Two Point Angular Correlation Function

I. Introduction

The Two Point Angular Correlation Function (TPACF) is a mathematical equation that has applications in many areas. In our project, the Two Point Angular Correlation Function is used as a way to measure the probability of finding an astronomical body at a given angular distance from another astronomical body. We were given working sequential code as a starting point. The sequential code took in two kinds of data to process. The first consisted of a set of valid data points measured from a region of space. These data points represented the spherical coordinates of actual astronomical objects in space. The number of data points that this set contained could range up to 97,178 data points. The second kind of data was randomly generated data. The randomly generated data sets contained coordinates similar to the actual data set, but with random distribution. The number of random sets used varied and could be set with an input parameter, but each set must contain at least the same amount of points as data points being used. It is not uncommon to run the TPACF program with as many as 100 random data sets.

$$w(\theta) = \frac{DD(\theta) - 2\frac{\sum DR(\theta)}{N} + \frac{\sum RR(\theta)}{N}}{\frac{\sum RR(\theta)}{N}}$$

Figure 1: TPACF Equation

The TPACF (listed above as w) is calculated according to the equation found in Figure 1. The three most computationally intensive parts of the equation are the computations of the correlation functions: DD, DR, and RR. DD is the auto-correlation of the actual data set. DR is the cross-correlation of the actual data set to a given random set. RR is the auto-correlation of the random sets. Note that there will be only one DD to calculate, but there will be multiple DRs and RRs to calculate. There will be as many DRs and RRs as there are random sets used. Each of the correlation functions consists of a series of dot products between coordinate pairs. Each of these calculations is an independent floating point operation that can be done in parallel. The results of these calculations are organized into histograms.

The sequential code first takes in the data set and random sets. Then it converts all the spherical coordinates from the input files to Cartesian coordinates. Next, the code finds

the bin boundaries that will be used by DD, DR and RR to generate the histograms. After that, the correlations functions (DD, DR, and RR) are calculated. Once the correlation functions have been computed, the results of the DR and RR calculations are summed to form a single DR and a single RR histogram. The final step is to calculate the TPACF for each bin boundary.

1.1 Parallelism

The high degree of parallelism in the TPACF makes it an interesting application to parallelize and run on the G80. The computations that could potentially be done in parallel are the conversion of data points from spherical to Cartesian coordinates, the dot product calculations involved in the correlation functions(DD, DR and RR), and the summations of DR and RR for use in the final calculation of the TPACF. The calculations to find DD, DR and RR are the most computationally intense part of TPACF. The calculations involved in finding these functions are independent both within the functions themselves and across functions. To exploit the parallelism within correlation functions, all the dot products done in a given correlation function can be spread across different threads. Given N elements in a set, there would be N*N calculations to do for the cross correlation functions (DRs) and approximately N*N/2 calculations to do for the autocorrelation functions (DD and RRs). The second kind of parallelism uses the fact that the correlations functions are independent from each other. So, the DD, DRs and RRs can be calculated in parallel. Given M random sets there would be M + 1 auto-correlations to calculate and M cross-correlations to calculate.

The greatest difficulty in parallelizing TPACF arises when the histograms are generated. Generating histograms is easy when there is one thread and things can be done sequentially. However, with multiple threads storing their results into the same histogram, there is a danger of losing the results of some threads due to scheduling race conditions. Each thread attempting to write into a shared histogram needs to load a value, increment that value, and store the value back. To prevent the loss of data, each thread uses its ID to tag its writes into the shared histograms. The threads attempt to read the data back after writing into the histogram. If the thread cannot find its own tag in the value it reads back, then it assumes that another thread overwrote its results and that it needs to reload the data, increment and then repeat the write/tag process. If there are too many conflicts, then there is a performance penalty due to re-execution of the write/tag and check process.

1.2 Precision Issues

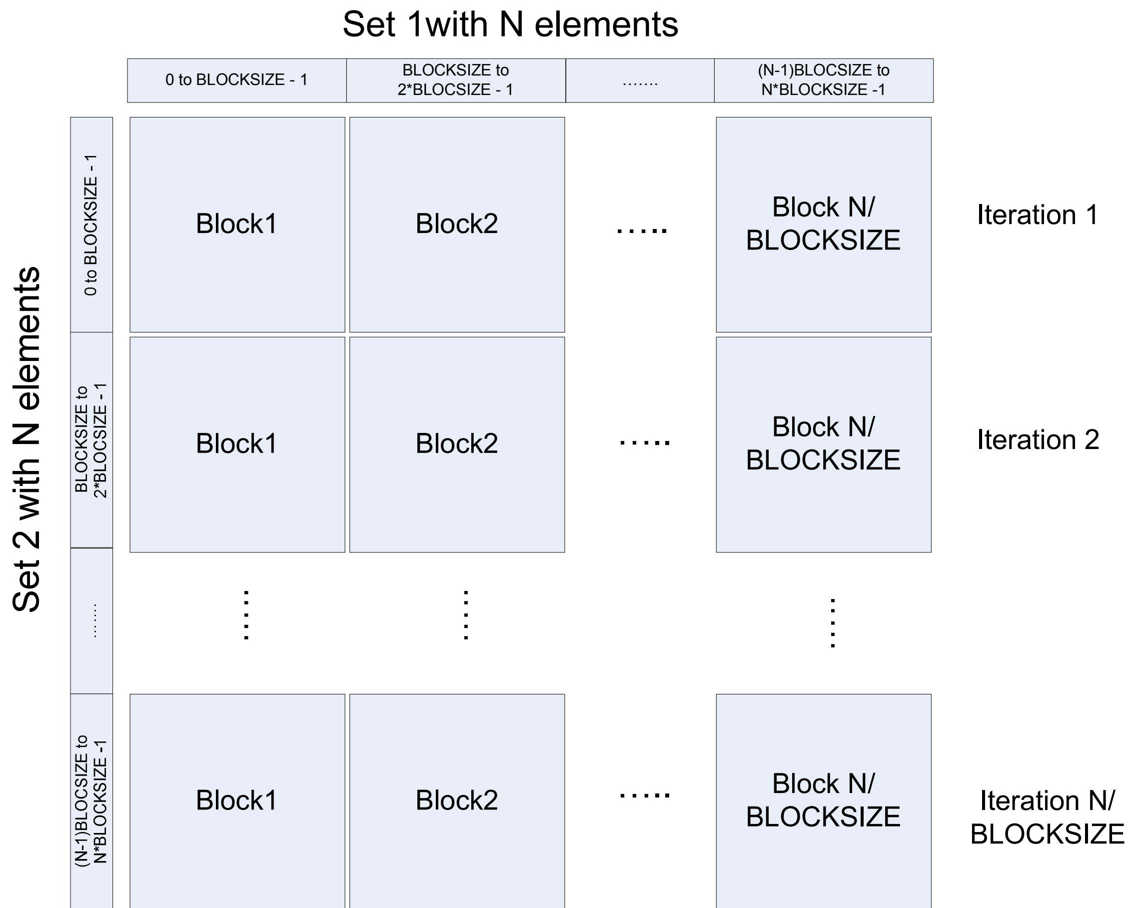
The CPU gold code provided by the sponsor calculates w for 30 different values of theta, organized in a logarithmic binning scheme ranging between 0.01 and 10000, with 5 bins per decade. It uses double-precision floating point to calculate which bin each distance value belongs in, i.e. which range of theta it fits in. Since CUDA does not yet support double-precision, we used single-precision floating point for these calculations. In calculating the distances, the smallest distances -- those which would belong to the 10 lowest bins -- cannot be differentiated in single-precision, and are treated as 0. As a result, these values are placed in the lowest bin instead of the proper locations. To correctly calculate the values for these lower bins requires 41 bits of fixed point precision, which exceeds what we have available. To resolve this issue, we decided to use only the top 20 bins out of the 30 that were in the gold code. The sponsor agreed to this.

In addition to the floating point precision limitations, the possibility exists for integer overflow in the bins of the histograms. Since the number of data points can range up to 97178, if too many data point pairs in any given correlation function fall into the same bin then the 32-bit bins can overflow. The DRs have the greatest danger of overflow. With 97178 data points there will be 97178*97178 data point pair dot products to place into a histogram per DR function. This is over double the maximum value that can be represented with an unsigned 32-bit integer. The problem is compounded once the DR histograms are summed together. We currently do not deal with this issue.

II. Design Overview

Our project had two implementations of TPACF for CUDA. One exploits the independence of the distance calculations within a set, with separate kernel calls for each set (DD, DR, RR's). The second version exploits the independence of the calculations of the sets themselves, resulting in just one kernel call, with a different block for each set.

Figure 2 below is the block overview of version 1, and figure 3 is version 3. The details of the algorithms are outlined in the implementation section below.



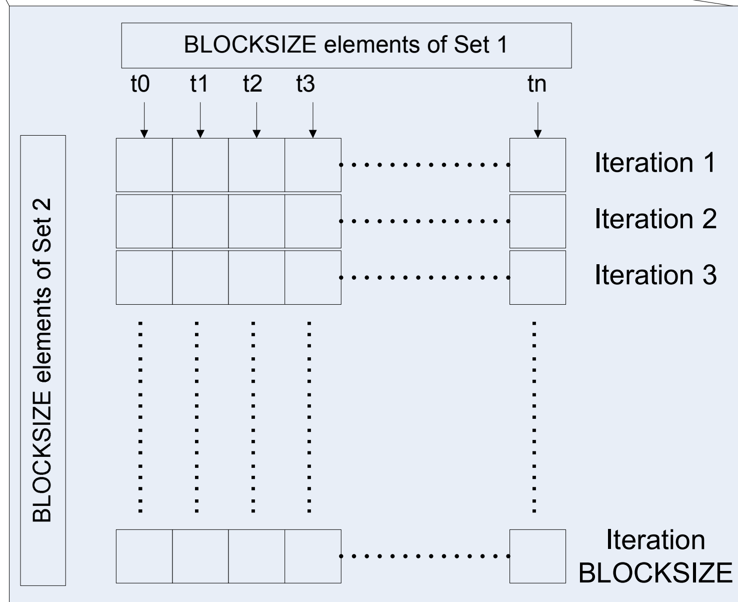


Figure 2: A single kernel call for version 1

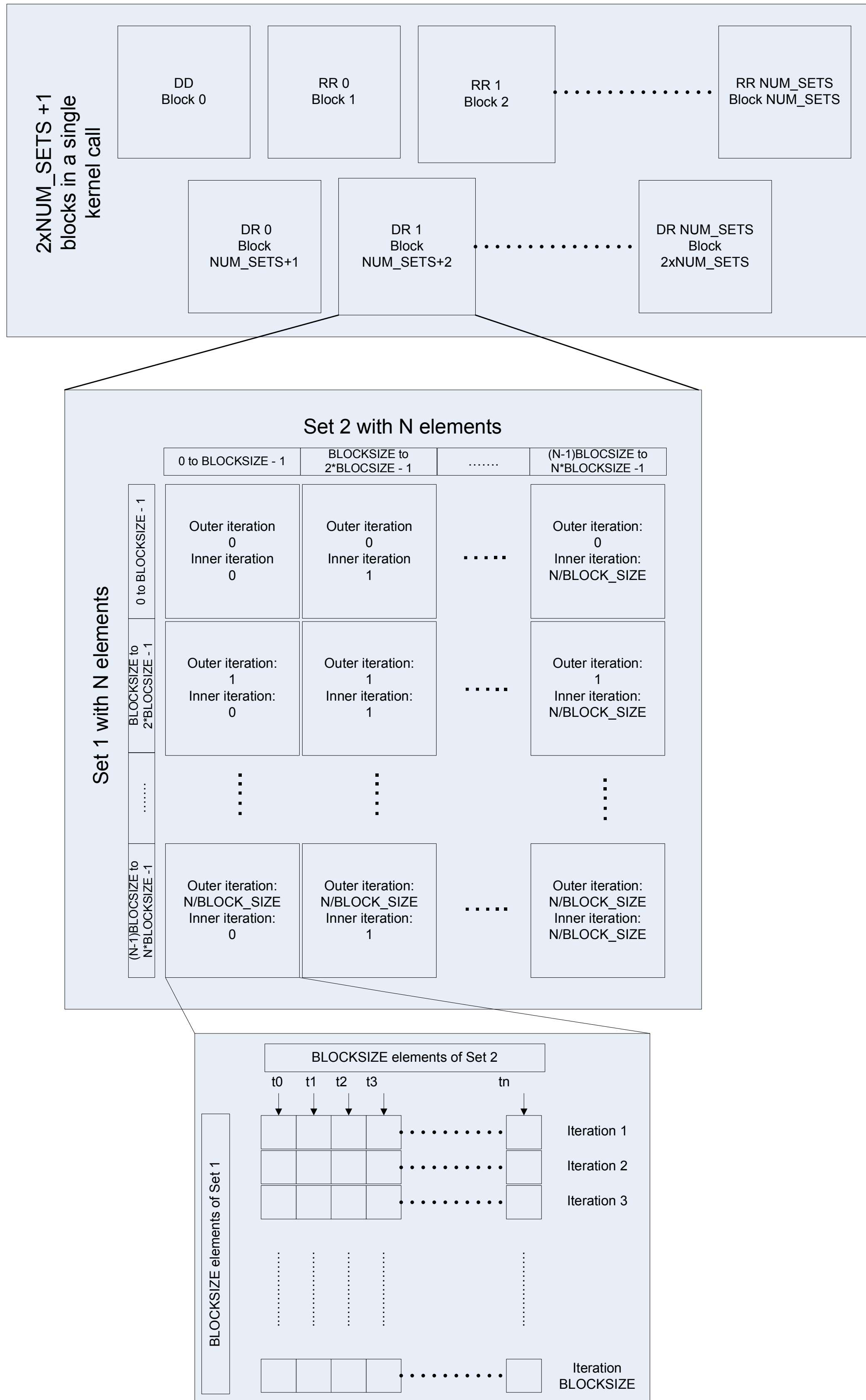


Figure 3: Kernel call for version 2 calculating all DD, DR and RR values

III. Implementation

Implementation Version 1
Our first attempt to implement this algorithm exploits parallelism within each correlation of a pair of sets (e.g. DD, a DR, or an RR). It consists of two kernel functions: one to take in the two sets to be correlated and perform the distance and binning calculations, and a second one to coalesce the results of the first function. For N random sets, there are 2N+1 sequential calls to the first kernel function -- one for DD, N for the DRs, and N for the RRs.

The calculation space is an MxM array, where M is the number of data points in each of the two input sets to be correlated. Each index corresponds to one distance calculation between a coordinate point in the first set and a coordinate point in the second set. Originally, version 1 gave each block BLOCK_SIZE calculations to do, i.e. a correlation between one point in the first set and BLOCK_SIZE ones in the second set. The only use for shared memory in this scheme is to store the point from the first set which is needed by all threads to correlate against each of their other points. This was modified into a two-dimensional scheme which makes better use of shared memory, and gives a marginal performance benefit.

Figure 2 shows how the input space is divided into blocks. Each block performs a BLOCK_SIZE x BLOCK_SIZE portion of the work. BLOCK_SIZE elements of Set 1 are loaded into shared memory for each block, with one element corresponding to each thread in the block. Then a loop that executes for BLOCK_SIZE iterations takes in one element of Set 2 per iteration, and each thread calculates the distance between the Set 2 element and its own Set 1 element. The corresponding bin number for this distance is then calculated. A histogram is generated in each half-warp, using the method described in the course slides to work around the lack of atomic instructions for writing into memory locations. Then, the warp histograms in each block are coalesced into one histogram per block using an indexing scheme designed to avoid bank conflicts.

The second kernel function is then called iteratively to coalesce the block histograms generated by the first kernel function. This second kernel uses a binary tree like the parallel scan algorithm of MP6, to generate a final histogram containing the total sum in each bin. This has log(x) iterations, where x is the number of histograms generated by the first kernel function.

The calculation of each correlation between a pair of sets consists of a call to the first kernel followed by several calls to the second. 2N+1 such correlations are done in sequence. Finally, w (theta) is calculated on the CPU for each bin, using the histograms resulting from the 2N+1 calls.

Implementation Version 2
Version 2 tries to exploit parallelism along the different DD, RR and DR calculations. It consists of single kernel call with a block for each DD, RR and DR calculation. For N random sets, the first N + 1 blocks calculate the auto-correlations (N random to self and 1

data to self). The remaining N blocks calculate the cross correlation of the data with the random sets. This is shown in Figure 3.

Each of the blocks must do calculations equal to the square of the number of elements for cross correlation and around half that for auto correlation. To perform these calculations each thread brings in a element from the first set into the shared memory. The distance calculations are performed for all the elements in the second set with the elements from the first set resident in shared memory.

This is implemented in the program using three for loops. The outermost loop iterates over all the elements in the first set and brings BLOCK_SIZE elements into shared memory. The middle loop iterates over all the elements in the second set and brings in one element into the register file for each thread, at a time. The innermost loop calculates the correlation of all the elements from the first set resident in shared memory with that threads element from the second set (resident in the register file).

Shared memory is used for holding the working set of the first set and the histograms. Initially it also held the second set elements. This was changed because the second set values are not shared across threads, so the second set elements can be held in a register. This brings down the usage of shared memory for the data down to 3kb, allowing us to put 16 histograms per warp.

After the kernel finishes execution we compute the omega values on the CPU. This requires summing up 200 histograms. There is still some parallelism in this but it scales only linearly with the size of the problem. Since the distance calculations increase quadratically with the number of elements we decided this would be a negligible cost.

Software Pipelining
To hide the latency of global memory accesses we tried using software pipelining. Instead of fetching the data to be used in the current iteration, we loaded the first set of data before entering the loop, and within the loop fetched the data for the next iteration. This was meant to make global memory accesses completely transparent at the cost of using more shared memory. However, when actually run, the overhead of implementing software pipelining and the cost of an extra 6k of shared memory was more than the benefit gained and the performance decreased slightly even when the histograms were disabled. Instead we opted to use that shared memory for increasing the number of histograms per warp, thereby decreasing our binning conflicts.

Version 2 Bank Conflicts
There are no bank conflicts when using the first set. When writing into shared memory each thread accesses a contiguous memory location. When reading from shared memory, all threads access the same memory location so the required word is broadcast.

The bank conflicts in histograms showed interesting behavior. Initially we declared the histograms as:

```
unsigned long warp_hists[NUM_BINS][NUM_HISTOGRAMS+1]
```

The +1 factor removes all bank conflicts as long as the number of histograms is a multiple of 16 or there are 2 or more histograms per warp. For 2 histograms per warp it boosts performance by 10%. Another alternative was to use the number of bins as the second array index. In our case of 20 bins this significantly reduces the bank conflicts and makes the performance the same as with the +1 case and actually better when there is only one histogram per warp. But the number of bins is a parameter to our program and we don’t want to force it to be a non power of 2, so we decided against this. Reducing the bank conflicts had a smaller effect as we increased the number of histograms per warp. Trying to reduce bank conflicts using 8 or 16 histograms per warp decreases he performance for a small amount. The other place where the histograms are accessed is when coalescing the histograms after finishing the calculations. Since this code is such a small part of our code, any optimizations we do for this would part would have a negligible effect.

Handling of Auto Correlation
When calculating the auto correlation for a dataset half the calculations are done twice (if all points are used). The gold code avoids all of these extra calculations. In the CUDA version we avoid most of the extra calculations by having the points loaded from the second set be based off the location of the points being used from the first set. This allows us to operate on chunks of data the same size as the square of the number of threads. By doing this we can eliminate most of our redundant calculations but we will still have some redundant calculations present.

Histogram Implementation
The histogram implementation followed the algorithm discussed in the lecture slides and recommended by NVIDIA (with the addition that the histograms were needed to be declared with the volatile keyword in addition to the shared keyword). Our

implementation added in an option to control the number of histograms allowed per warp. This allowed us to use 1, 2, 4, 8 or 16 histograms per warp. 32 histograms per warp or a histogram for each thread requires 20kb of shared memory which is more than what we have. This histogram algorithm works worst when multiple threads in the same warp want to write into the same bin. That is actually the most common case for our program. Because of this the option to increase the number of histograms per warp proved very helpful.

To eliminate binning conflicts we originally attempted a completely different binning technique. Ideally we want each thread to have its own histogram but we don't have enough shared memory to do that. We tried to keep a histogram for each thread in the register file in variables of type char. We then added these values to the actual histogram kept in shared memory since the range is very limited in the per thread histograms we were using. To avoid using local memory since the arrays for histogram cannot be kept in registers, we tried packing multiple histogram bins into several variables of type unsigned long. We got this histogram scheme to work under emulation but we were unable to make it work on the device. We were able to isolate the problem to an "Unspecified launch failure".

IV. Verification

For version one of our code we tested with up to 16384 data points and 10 random sets. For version two of our code we tested with up to 24576 data points and 7 random sets, as well as 8192 data points and 95 random sets.

The main obstacle in further testing with larger data sets was kernel crashes for long kernel calls. Kernels are killed after approximately eight seconds. The single precision issues listed in section 1 were considered in deciding the acceptable accuracy for the final implementations. Even after adjusting the gold code to single precision our final results did not perfectly match the gold code results. This is due to the fact that the CPU uses internal registers that are larger than 32-bits to store intermediate floating point results. For acceptance testing, we considered 10% error in lower 3 bins for TPACF or w(theta) to be acceptable. For the upper bins (which contain the must points and hence normalize the calculations), we considered 1% error to be acceptable.

V. Performance

Of the two kernel functions in Version 1, the first one does the majority of the work and takes up most of the execution time of the program. Version 1 serializes different sets of calculations, and parallelizes only across individual sets. This gives more resources to work on each set of calculations, but does not exploit the fact that sets of calculations are also independent and can be parallelized. Version 1 also makes relatively little use of shared memory, reading values from global memory more often than Version 2 does. In general, this version of the code is approximately 1.5 to 2 times faster than the CPU gold code.

Several attempts were made to backport Version 2's optimizations into equivalent portions of Version 1's code. In general, these optimizations had little or no effect on

overall performance, though some of them moved the bottleneck from writing to the warp histograms, to coalescing the histograms in a block (or vice versa). The ideas tried included multiple histograms per warp, using shared memory more efficiently to reuse data points, and removing bank conflicts by making array strides non-power-of-two.

In implementation 2, we surpassed our original performance goals (of at least 10x). Although we have not been able to run the full dataset due to the time limitations of kernel invocations, our results for shorter calls have been accurate (according to the previous verification standards), so we believe the algorithm to be accurate.

Figure 4 shows our speedup compared to the most optimized gold code provided by the sponsor when we keep the dataset size at 2048 and vary the number of blocks (which is itself dependant on the number of random sets used). Our best speedup is around 44x is in this case.

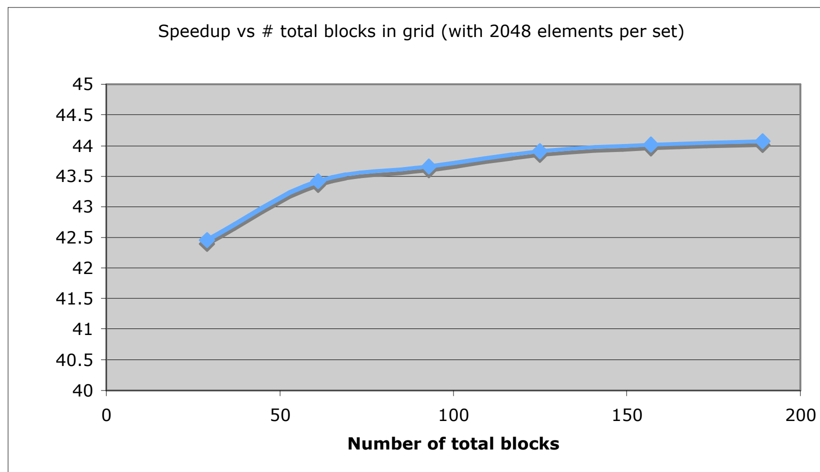


Figure 4: Speedup vs Grid size

Only those grid sizes that use the largest percentage of the SM's available on the chip are shown. For the G80, these are those that are closest to multiples of 32. Although the G80 has 16 SM's, and those grid sizes closest to multiples of 16 should be the "sweet spots", the speedup was consistently better when the grid size was close to a multiple of 32. Although using other grid sizes doesn't hurt performance, it does not take as full advantage of the SM's available, and therefore the speedup is not as large.

Figure x2 shows the speedup when the number of random sets is kept at 7 blocks, and the dataset size is varied. In this case, the size of the dataset results in an exponential expansion of the processing time. Therefore, although the maximum dataset size is 97178, we were only able to process up to 24k elements, due to the kernel crashing on larger dataset sizes. 7 random sets means that there are only 15 total blocks available. Although this results in a smaller speedup than if 31 blocks were being used, it allowed us to test larger dataset sizes. Again, the more elements available, the more speedup shown, although the difference is not as pronounced as varying the grid size.

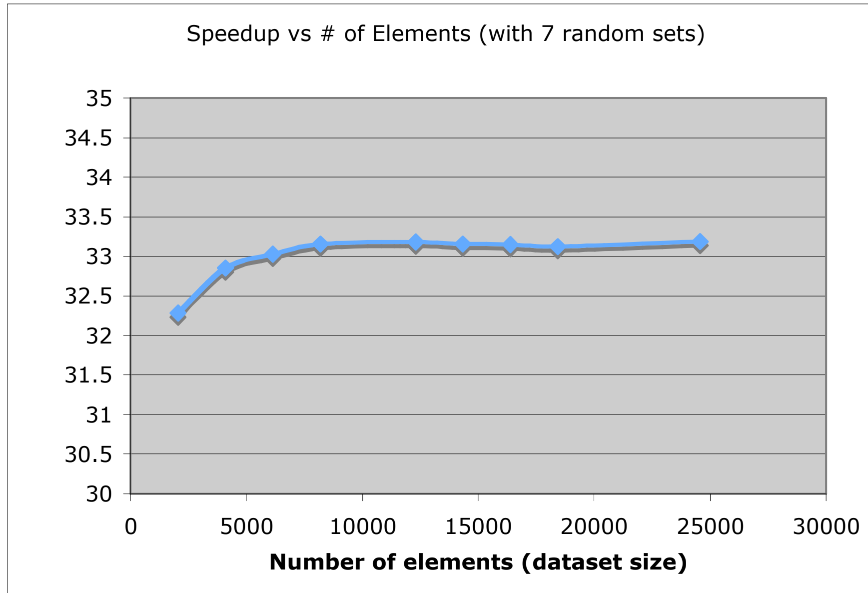


Figure 5: Speedup vs Element size

In all, our best speedup was for a random set size of 95 (giving us 191 available blocks) and a data set size of 8192, for a total speedup of 46.535181x.

These numbers are significantly better than those presented in the presentation. This is due to the modifications mentioned in the implementation of version 2: putting the random sets in registers instead of shared memory, thereby letting us use more shared memory for the warp histograms which let us have 16 histograms per warp; removing some unnecessary if statements in our innermost loop; making all integer calculations unsigned; and reducing the number of redundant calculations for the autocorrelation portions of the algorithm.

VI. Conclusion

Overall, the TPACF algorithm has turned out to be well suited to CUDA, with max speedup greater than 45x compared to a sequential implementation. Because not only are the results of the distance calculations within a set independent of each other (which version 1 exploits), the sets themselves are independent (which version 2 exploits). As mentioned before, the largest factor preventing further speedup is the lack of an atomic increment instruction, which leaves room for race conditions when updating the histogram. We were forced to take this into account in our design, but if that were not an issue a better solution may have been possible.

Another issue, which we were forced to deal with, was that due to the large number of input values needed, in order to perform efficient calculations we were forced to push the shared memory usage and register usage to the limits. Future hardware revisions with more resources may remedy this problem as well as that of the loss of accuracy due to single precision values.

Nevertheless, given more time we feel that further optimization would have been possible. Due to the logarithmic nature of the bins, most of the binning conflicts occurred on the higher bins. If we were to split these up, by adding more bins with smaller ranges, we would most likely reduce bin conflicts.

Another possibility, which we did not implement, is using texture memory to implement a lookup table for deciding which bin a distance calculation falls into. Currently we have a binary search that can take several iterations. A lookup table, although it would take up a fair amount of texture memory space, would make the decision take constant time.

Lastly, we followed the binning algorithm recommended in the lecture slides. Although we attempted other algorithms, we were unable to get them to work when running on hardware. It may be that these others would be more efficient if we had enough time to get them to work.

In terms of the process involved with developing for CUDA, our largest complaint was the lack of information related to kernel crashes. In general, the string returned by the "check error" function gave little to no information (i.e.: unspecified launch failure tells us nearly nothing).

Another issue was that it was often difficult to tell when we were approaching the limits of the hardware in terms of resource usage. It would be helpful if the compiler or some other tool were capable of telling us this.

Overall though, we consider the project to have been a success. We ported the algorithm to CUDA successfully (considering the CUDA single-point precision and kernel time limitations), showed significant speedups, and in so doing showed that correlation and histogram algorithms in general are well suited for CUDA.