NCSA Technical Report, 2009

Investigating Application Analysis and Design Methodologies for Computational Accelerators

Volodymyr Kindratenko, Robert Brunner, Guochun Shi, Dylan Roeh, Austin Martinez
University of Illinois at Urbana-Champaign

1 Introduction

For many scientific and engineering applications the computational needs are growing faster than the capabilities of available computing hardware. Due to the technological difficulties in achieving higher clock speed of modern microprocessors—until recently the main performance improvement source—computational scientists are turning their attention to application hardware accelerators, among other, as means to satisfy the required computational needs. Supercomputer designers are also exploring the use of computational accelerators and multi-paradigm hybrid architectures as methods to further enhance performance. With AMD and Intel adding support for accelerators, next-generation high performance computing (HPC) cyberresources will likely include such acceleration technologies as integral parts of computer systems.

The impact of these hardware design trends on scientific applications and the investment required to utilize these resources is not fully understood in the scientific computing community. While next-generation computing architectures offer great *potential* performance, the execution models, software architectures, and development processes that are required to realize that potential currently differ dramatically from existing computational architectures. These new architectures require different methods for application analysis, hardware selection, software development, application adaptation, and performance optimization. Methods and practices for these new architectures are not well-developed and existing knowledge and experiences are fragmented across small groups of early adopters. While numerous exemplary applications have been implemented on accelerator architectures—with varying success—the scientific computing community has yet to take a broad view on the problem and to formulate formal procedures and methods for application development when considering accelerator technologies.

In this report, we document an exploratory investigation to understand the impact of accelerator technologies on scientific and engineering codes and to quantify the efforts and requirements necessary to implement these codes on the newly emerging accelerator technologies. We implement a commonly used algorithm on several accelerator architectures and, in doing so, develop guidelines and recipes that other researchers can adopt when porting their own applications to similar accelerator-based architectures. The chosen accelerators are NVIDIA GPUs, IBM Cell Broadband Engine, and Nallatech H101 FPGA accelerator, their characteristics are listed in Table 1. The chosen algorithm is the two-point angular correlation function (TPACF) as applied in the field of astronomy. Correlation analyses are a common tool from the field of spatial statistics, and thereby impact a wide range of scientific disciplines.

The report is organized as follows. We first introduce the mathematics behind the two-point angular correlation function. We then briefly discuss its implementation on a conventional microprocessor followed by a detailed analysis and implementation descriptions for FPGA, GPU, and Cell architectures. We conclude the report with an analysis of design methodologies for different accelerator architectures.

Processor type	*# of cores*	*Frequency*	*Attached memory*	*Attached memory bandwidth*	*On-chip memory*	*Peak GFLOPS*
AMD Opteron dual-core 2216	2	2.4 GHz	4 GB DDR2	10.7 GB/s	2x1 MB (L2 cache)	19.2 (DP)
IBM Cell/B.E.	9	3.2 GHz	512 MB XDR	25.6 GB/s	512 KB (L2) + 8 x 256 KB (SPE LS)	230 (SP) 21 (DP)
IBM PowerXCell 8i	9	3.2 GHz	8 GB DDR2	25.6 GB/s	512 KB (L2) + 8 x 256 KB (SPE LS)	230 (SP) 108.5 (DP)
NVIDIA Quadro FX 5600	128	1.35 GHz	1.5 GB GDDR3	76.8 GB/sec	16 x 16 KB (per multiprocessor)	346 (SP)
NVIDIA GeForce GTX 280	240	1.3 GHz	1 GB GDDR3	141.7 GB/s	30 x 16 KB (per multiprocessor)	933 (SP) 77.8 (DP)
Xilinx Virtex-4 LX100	N/A	200 MHz	16 MB SRAM	6.4 GB/sec	0.5 MB (BRAM)	6.4 (DP) 20 (SP)
			512 MB SDRAM	3.2 GB/sec		

Table 1. Characteristics of processors and accelerators used in this study.

2 Two-point angular correlation function

Correlation functions are used extensively within the astronomy community to characterize the clustering of extragalactic objects. The two-point correlation function encodes the frequency distribution of separations between coordinate positions in a parameter space, as compared to randomly distributed coordinate positions across the same space (Figure 1). In astronomy applications, a common coordinate choice is the angular separations, θ, on the celestial sphere, which can be used to measure the angular two-point correlation function, which we will denote here as $\omega(\theta)$. Qualitatively, a positive value of $\omega(\theta)$ indicates that objects are found more frequently at angular separations of θ than would be expected for a randomly distributed set of coordinate points (i.e., a *correlation*). Similarly, $\omega(\theta)=0$ codifies a random distribution of objects, and $\omega(\theta)<0$ indicates an unexpected paucity of objects at separations of θ (i.e., an *anti-correlation*).

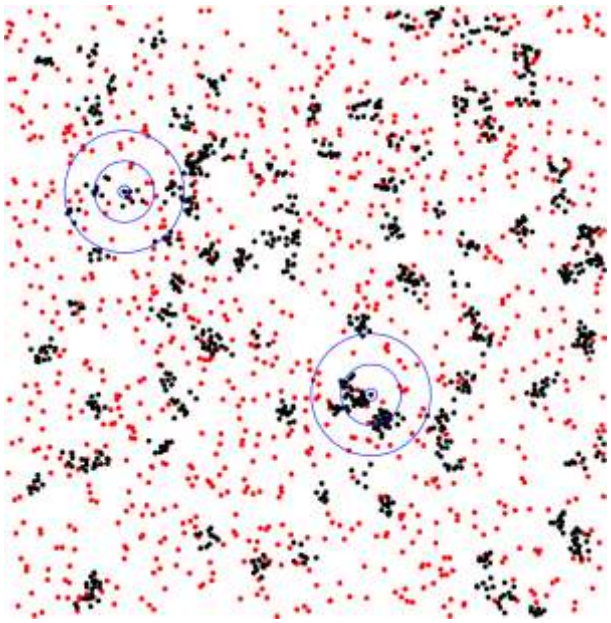

Figure 1. Two-point angular correlation function, $\omega(\theta)$, is the frequency distribution of angular separations θ between celestial objects in the interval (θ, $\theta+\delta\theta$). $\omega(\theta)=0$ for randomly distributed (red) points and $\omega(\theta)>0$ for clustered (black) points. $\omega(\theta)$ can vary as a function of angular distance: $\omega(\theta)=0$ for random (red) points on all scales whereas $\omega(\theta)$ is larger on slammer scales for clustered (black) points.

Estimating angular correlation functions generally requires computing histograms of angular separations between a particular set of positions in a data space. The positions in question might be the set of data points themselves, which we will denote $DD(\theta)$, or a set of points that are randomly distributed in the same space as the data, which we will denote $RR(\theta)$. Similarly the distribution of separations between the data sample and a set of random

points, which we will denote $DR(\theta)$, can be calculated. Once such distributions are known, $\omega(\theta)$ is estimated as in (1):

$$\omega(\theta) = \frac{DD(\theta) - 2DR(\theta) + RR(\theta)}{RR(\theta)} \tag{1}$$

Naively, calculation of the separation distributions (*DD, DR, RR*) for N_D total points is an $O(N_D^2)$ problem, as it requires computing distances between all possible pairs of points in the data space. Additionally, as the variance of each of the separation distributions diminishes with an increase in the number of points sampled, using a random sample that is n_R times larger than the dataset, and then renormalizing by dividing out by the factor of n_R, is recommended. This guarantees that the finite size of the random sample introduces a contribution to the variance that is n_R times smaller than the contribution from the data sample itself. To ensure the random points introduce fractional statistical imprecision compared to the natural limitations of the data, the random sample is usually constructed to contain $n_R \sim 100$ times as many coordinate positions as the dataset.

Computing the distribution of *all* separations for a random sample that is n_R times larger than a dataset increases calculation complexity by a factor of n_R^2. As modern astronomical data sets can contain many millions of positions, complexity can grow rapidly. One might therefore prefer to create n_R unique random samples of comparable size to the dataset, and then average the separation distributions over these individual realizations, thus reducing the complexity introduced by sampling across the random realizations to n_R. Fortunately, statistical precision is not reduced by such an approach. Equation 1 can then be written:

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

where n_R is the number of sets of random points.

The source of computational complexity for the TPACF comes from computing the $DD(\theta)$, $RR(\theta)$, and $DR(\theta)$ histograms of angular separations, which requires computing distances between all possible pairs of points in the datasets. Algorithms for computing histograms of angular separations are shown in Figure 2 and Figure 3 where, "binmap" refers to a subroutine that finds the bin in which the given angular separation θ falls. Astronomical measurements are usually made in a spherical coordinate system, with the coordinate positions expressed as Right Ascension and Declination (i.e., latitude and longitude) pairs. The separation, θ, between any two positions *p* and *q* in such a coordinate system can be determined by first converting the spherical coordinates to Cartesian coordinates, and computing θ:

$$\theta = \arccos(p \cdot q) = \arccos(x_p x_q + y_p y_q + z_p z_q) \tag{3}$$

The binning schema implemented by astronomers is typically logarithmic, as clustering patterns can be important in extragalactic astronomy across a wide range of angular scales. Each decade of angle in the logarithmic space is divided equally between *m* bins, meaning that there are *m* equally-logarithmically-spaced bins between, for example, 0.01 and 0.1 arcminutes. The bin edges are then defined by $10^{p/m}$, where $p=-\infty,\ldots,-1,0,1,\ldots+\infty$, and the bin number for angular separation θ can be found by projecting logarithm of θ onto an interval of integer values, from 0 to *M*, that define bins with boundaries $10^{p/m}$:

$$bin = \mathrm{int}\left[m(\log_{10}\theta - \log_{10}\theta_{min})\right]. \tag{4}$$

where θ_{min} is the smallest angular separation that can be measured and M is the total number of bins.

Algorithm 1: autocorrelation (DD and RR bin counts)

Input: 1) set of Cartesian coordinates of N points x_1, .., x_N on the celestial sphere; 2) for each point x_i, jackknife sample number, $1 \le s_i \le K$, from which x_i is removed; 3) set of M bins given by their boundaries: $[\theta_0, \theta_1)$, $[\theta_1, \theta_2)$, .., $[\theta_{M-1}, \theta_M)$;

Output: for each bin in every jackknife sample, the number of unique pairs of points (x_i, x_j) for which the angular separation $\theta(x_i,x_j)$ is in the respective bin;

```
1.  for i=1,..., N-1 do
2.      for j=i+1,..., N do
3.          l←binmap(θ(x_i,x_j), θ)
4.          for k=1,..., K do
5.              if s_i ≠ k do B_{k,l}← B_{k,l} +1
```

Figure 2. Autocorrelation algorithm.

Algorithm 2: cross-correlation (DR bin counts)

Input: 1) 2 sets of Cartesian coordinates of N points x_1, .., x_N and y_1, .., y_N on the celestial sphere; 2) for each point x_i, jackknife sample number, $0 \le s_i \le K-1$, from which x_i is removed; 3) set of M bins given by their boundaries: $[\theta_0, \theta_1)$, $[\theta_1, \theta_2)$, .., $[\theta_{M-1}, \theta_M)$;

Output: for each bin in every jackknife sample, the number of unique pairs of points (x_i, y_j) for which the angular separation $\theta(x_i,y_j)$ is in the respective bin;

```
1.  for i=1,..., N do
2.      for j= 1,..., N do
3.          l←binmap(θ(x_i,y_j), θ)
4.          for k=1,..., K do
5.              if s_i ≠ k do B_{k,l}← B_{k,l} +1
```

Figure 3. Cross-correlation algorithm.

Theoretically, each possible angular separation lies in a unique bin, and $\omega(\theta)$ can thus be uniquely determined for any distribution of points. Angular coordinates, however, as measured by modern astronomical surveys, are typically precise to ~0.1 arcseconds. The definitions of the bin edges are absolute; but the θ values have some built-in tolerance. Expressing θ values to different numbers of decimal places, therefore, can cause separations to drift between bins, affecting an estimate of $\omega(\theta)$, but *not rendering that estimate incorrect*. Any differences in the estimates of $\omega(\theta)$ that are introduced by imprecision in measured coordinates are, in most instances, completely undetectable, as variations in the random samples used to estimate $DR(\theta)$ and $RR(\theta)$ will usually dominate this numerical imprecision.

Note that the binning schema described above requires the calculation of *arccos* and *log* functions, which are computationally expensive. In practice other algorithms can be used. Based on the observation that if only a small number of bins are required, a faster approach is to project the bin edges to the pre-arccosine "dot product" space and search in this space to locate the corresponding bin. The values of bin edges, $10^{p/m}$, can be converted to the "dot product" space by computing their cosine:

$$binedge_p = \cos(10^{\log_{10}\theta_{min} + p/m}),\ p = 0, \ldots M \quad (5)$$

Since the bin edges are ordered, an efficient binary search algorithm can be used to quickly locate the corresponding bin in just $\log_2 M$ steps. However, in case when a significant bias is present with a significant fraction of values falling into just a few bins at the end or the beginning of the bin range, the binary search algorithm may be less efficient than a straightforward sequential search.

Error estimates for two-point angular correlation function can be computed using jackknife re-sampling technique in which correlation values are systematically recomputed leaving out one observation at a time from the sample set. In practice this is implemented by labeling each observed object with the sample number, s, $1 \le s \le K$, from which it is removed and updating only those histograms of angular separations, $DD(\theta)$ and $DR(\theta)$, that do not contain points from sample s. Thus, K independent estimates of $\omega(\theta)$ are computed and their variance provides a robust error estimate for the TPACF.

3 Conventional processor implementation

Computing histograms of angular separations, $DD(\theta)$, $RR(\theta)$, and $DR(\theta)$ for some number of jackknife samples is implemented in the C programming language as a shown in Figure 5. The kernel, called `doHistogram`, is invoked with two datasets, `data1` and `data2`, each consisting of `n1` and `n2` points, respectively, `nbins` bin boundaries stored in `binb` array, and the number of jackknife samples used, `njk-1`. As its output, this function computes `njk` histograms that are stored as `data_bins`, which are updated with the angular separations for the points from `data1` and `data2`. Separate arrays are used for $DD(\theta)$, $RR(\theta)$, and $DR(\theta)$ histograms. When the subroutine is called to compute $DD(\theta)$ or $RR(\theta)$ histograms, `doSelf` flag is set to 1, indicating that only one dataset is supplied and the amount of calculations can be cut in half. The kernel is first invoked to compute $DD(\theta)$ histograms and then is invoked $2n_R$ times to compute $RR(\theta)$ and $DR(\theta)$ histograms for n_R datasets with random data. Finally, Equation 2 is applied to compute $\omega(\theta)$.

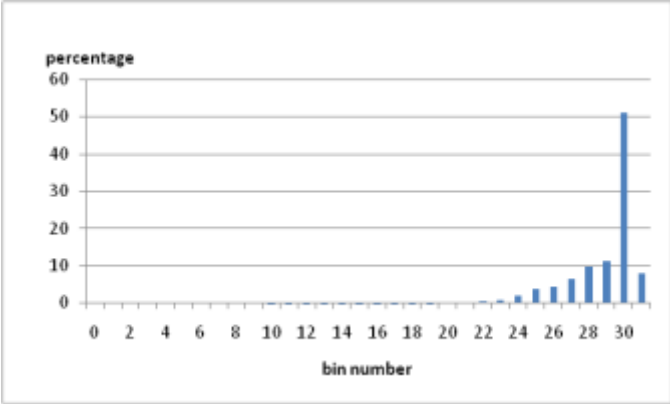

Figure 4. A typical histogram distribution shows a significant bias in bin counts for large angular separations.

In the reference C implementation the two algorithms presented in Section 2 are merged into a single C subroutine that executes either in auto- or cross-correlation fashion depending on the input parameters. We also compared three different approaches to compute bin mapping: direct formula that involves computing *arccos* and *log* functions, binary search, and sequential *waterfall* search that starts from one end of the bin boundaries and sequentially tests them until the right bin is found. We found that the direct formula is the slowest whereas the waterfall search is the fastest. Given the nature of the binned separations, this result is not surprising as there is a significant skew in bin counts towards larger angular separations (Figure 4). As a result a waterfall search on average requires less than $\log_2 M$ comparisons, as in the binary search, to find the correct bin.

Our reference C implementation is compiled with Intel C compiler and `-fast` optimization level. Care is taken to optimize the code for best performance. For reference, we provide an example run of the application. The dataset and random samples we use to calculate two-point angular correlation in this

work are the sample of photometrically classified quasars and the random catalogs first analyzed by (2) (Figure 6a). We use 100 random samples (n_R=100). The dataset and each of the random realizations consist of 97,178 points (N_D=97,178). A binning schema with five bins per decade (m=5), θ_{min}=0.01 arcminutes, and θ_{max}=10,000 arcminutes. Thus, angular separations are spread across 6 decades of scale and require 30 bins (M=30). Errors are estimated by removing 10% of point from the dataset and re-computing statistics 10 times (Figure 6b). It takes over 13 hours to execute the code on a single 2.4 GHz AMD processor.

```
01 void doHistogram(struct cartesian *data1, int n1, struct cartesian *data2, int n2,
02                 int doSelf, long long **data_bins, int nbins, double *binb, int njk)
03 {
04    if (doSelf) { n2 = n1; data2 = data1; }  // setup pointers for Self-compute mode
05    for (i=0; i<((doSelf)?n1-1:n1); i++) {  // loop over points in the first dataset
06       double xi=data1[i].x;  double yi=data1[i].y;    // retrieve point from
07       double zi=data1[i].z;  int jk=data1[i].jk;      // first dataset to registers
08       for (j=((doSelf)?i+1:0); j<n2; j++) {  // loop over points in the 2nd dataset
09           double dot=xi*data2[j].x+yi*data2[j].y+zi*data2[j].z;    // compute dot P
10           int indx, k=nbins;                                  // find bin it belongs to
11           if (dot>=binb[0]) indx = 0;       // eliminate those outside of the range
12           else { while (dot > binb[k]) k--;  indx = k+1; }     // sequential search
13           for (l = 0; l < njk; l++) if (l != jk) data_bins[l][indx] += 1; // update
14       }
15    }
16 }
```

Figure 5. The C implementation of the kernel that computes histograms of angular separations.

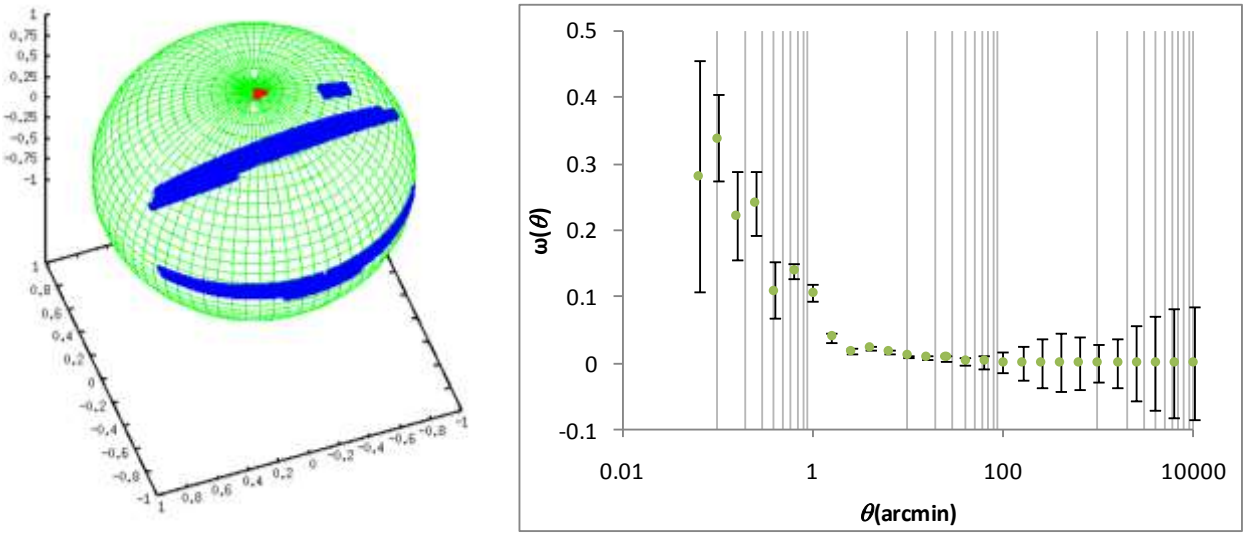

Figure 6. **a)** Dataset. **b)** Two-point angular correlation function.

4 FPGA implementation

Our FPGA implementation of the TPACF is designed to run on the Nallatech H101-PCIX FPGA accelerator board. The Nallatech H101-PCIX FPGA accelerator board used in this study includes a Xilinx Virtex-4

LX100 FPGA chip, 16 MB SRAM, and 512 MB SDRAM (Figure 7) (3). The card connects with the host system via a 133 MHz PCI-X interface. We use Nallatech's field upgradeable systems environment (FUSE) API (4) to interface with the card, and we develop applications for the accelerator card using the DIMEtalk application development environment (5). To describe the connectivity of various hardware modules used by the application, such as PCI-X interface and off-chip memory banks, we use DIMEtalk's network editor. The actual FPGA design is implemented as a DIMEtalk network module using DIME-C programming language. The translation from DIME-C source code to VHDL is performed by Nallatech's DIME-C C to VHDL Function Generator. DIMEtalk System Design generates VHDL code and user constraints files from the DIMEtalk network. Finally, the Xilinx Integrated Software Environment (ISE) (6) synthesized and implemented the design for the FPGA.

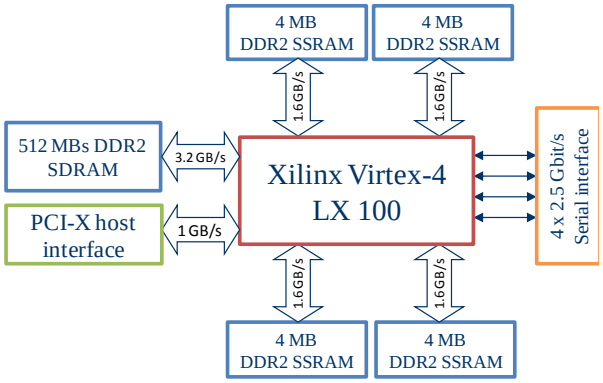

Figure 7. Nallatech H101-PCIX FPGA accelerator board architecture.

4.1 Suitability of the Nallatech H101 for the TPACF

We first analyze the suitability of the H101 accelerator for implementing the computational kernels presented in Section 2. We apply the RC amenability test (RAT) (7), which consists of three parts: throughput, numerical precision, and resources usage. The RAT produces an estimate of expected speedup, if any, and may give an early indication if the kernel will not fit on the FPGA.

4.1.1 Throughput

In the RAT, an application's predicted performance is defined as the sum of the host-FPGA communication time and the FPGA kernel execution time, ignoring the time required for reconfiguration and setup. A very basic algorithm description is needed to apply the RAT throughput test. We present the algorithm and system parameters necessary to compute the throughput in Table 2, while the computed throughput parameters are shown in Table 3.

Application dataset parameters. We consider a specific case in which the algorithms presented in Section 2 are executed once on a dataset consisting of 32,768 points. The autocorrelation algorithm (**Alg1** in tables below) requires one such dataset, whereas the cross-correlation algorithm (**Alg2**) requires two such datasets. Thus the number of input elements ($N_{elements}$, input in Table 2) for these two algorithms are 32,768 and 65,536 respectively. Each element is described by its three Cartesian coordinates and by a number that specifies the element's jackknife sample. We consider the most general case in which the Cartesian coordinates are stored as double-precision (64-bit) floating-point numbers and the jackknife sample number is stored as a 64-bit integer. Thus, each element requires 32 bytes of storage space ($N_{bytes/element}$, input). We assume that 32 bins and 10 jackknife samples are required, which are typical values, resulting in 320 output elements ($N_{elements}$, output), which are 64-bit integer numbers ($N_{bytes/element}$, output). We assume that bin boundaries are pre-computed and stored in the FPGA's distributed memory as part of the FPGA kernel initialization.

Hardware communication parameters. The Nallatech H101 card is connected to the host system via a 133 MHz PCI-X bus that has a theoretical maximum bandwidth of 1 GB/s ($throughput_{ideal}$). The fraction

of ideal throughput that is performing useful communication—α parameters for read and write in RAT's terminology—is taken from (7) and is based on micro-benchmarks provided by the RAT developers.

Kernel computation parameters. In the cross-correlation algorithm (**Alg2**), each point from the first dataset is evaluated against each point from the second dataset. Each such computation involves 3 multiplications, 2 additions, and $\log_2 32=5$ comparison operations (when bin mapping is implemented as a binary search), or 10×32,768 operations in total per single element from the first dataset ($N_{ops/element}$). For the auto-correlation case (**Alg1**), the number of calculations per element is cut roughly by a factor of 2. We assume that we will be able to place just one instance of each of the two algorithms on an FPGA and that we will be able to fully pipeline the computations, thus, we expect to perform 10 operations per cycle ($throughput_{proc}$), while running our FPGA design at 100 MHz (f_{clock}), which is a conservative assumption for an FPGA capable of running at 500 MHz.

Software implementation. Our most efficient software implementation of these two kernels using the aforementioned input parameters on a 2.4 GHz AMD Opteron processor executes in 17.8 and 40.5 seconds for **Alg1** and **Alg2**, respectively.

		Alg1	**Alg2**
Application Dataset Parameters			
$N_{elements}$, input	elements	32,768	65,536
$N_{bytes/element}$, input	bytes/element	32	
$N_{elements}$, output	elements	320	
$N_{bytes/element}$, output	bytes/element	8	
Nallatech H101 Communication Parameters (7)			
$throughput_{ideal}$	MB/s	1,000	
α_{write}	0<α<1	0.147	
α_{read}	0<α<1	0.001	
Kernel computation parameters			
$N_{ops/element}$	ops/element	~163,840	327,680
$throughput_{proc}$	ops/cycle	10	
f_{clock}	MHz	100	
Software Implementation Parameters			
t_{soft}	sec	17.8	40.5
N_{iter}	iterations	1	

Table 2. RC amenability test parameters.

Performance prediction. Table 3 contains performance predictions based on the methodology from (7) using the algorithm and system parameters from Table 2. As an example for **Alg1**:

T_{read} = (32,768 elements × 32 bytes/element) / (0.147 × 1,000 Mbytes/sec) ≈ 0.007 seconds

T_{write} = (320 elements × 8 bytes/element) / (0.001 × 1,000 Mbytes/sec) ≈ 0.003 seconds

T_{comm} = 0.007 seconds + 0.003 seconds = 0.01 seconds

T_{comp} = (32,768 elements × 163,840) / (100 MHz × 10 ops/cycle) ≈ 5.369 seconds

$t_{RC_{SB}}$ = 1 iteration × (0.01 seconds + 5.369 seconds) = 5.379 seconds

speedup = 18.4 seconds / 5.379 seconds ≈ 3.27

$util_{comm_{SB}}$ = 0.01 seconds / (0.01 seconds + 5.369 seconds) ≈ 0.0019

$util_{comp_{SB}}$ = 5.369 seconds / (0.01 seconds + 5.369 seconds) ≈ 0.9981

To summarize, the RAT throughput test indicates the algorithms are compute-bound ($util_{comp_{SB}}$>> $util_{comm_{SB}}$) and that we should expect 3.3× to 3.8× speedup when implementing the two algorithms on the Nallatech H101 FPGA accelerator board.

	Alg1	**Alg2**
t_{comm} (sec)	0.01	0.017
t_{comp} (sec)	5.369	10.737
$util_{comm_{SB}}$	0.0019	0.0016
$util_{comp_{SB}}$	0.9981	0.9984
$t_{RC_{SB}}$ (sec)	5.379	10.754
speedup	**3.3**	**3.8**

Table 3. RC amenability test predictions.

4.1.2 Numerical precision

Angular coordinates, as measured by modern astronomical surveys, are typically precise to ~0.1 arcseconds. The difference between the two smallest bin edges, 0.01 and 0.1 arcminutes, when converted to radians, is $6 \cdot 10^{-12}$. Thus, twelve decimal digits (40 bits of the mantissa) are sufficient to provide the typical required precision. DIME-C, however, does not support fixed-point arithmetic, while performing calculations with single-precision floating-point arithmetic will not be sufficient to cover angular separations below 1 arcminute. Double-precision arithmetic, which is supported by DIME-C on H101 FPGA accelerators, will be sufficient. Thus, the numerical precision requirements of the algorithms are within the capabilities of the hardware/software toolkit.

4.1.3 Resources

We also evaluate the availability of basic FPGA and system resources, such as on-board and on-chip RAM, hardware multipliers, and random logic elements, which are necessary to implement the algorithms. DIME-C uses IP cores generated by Xilinx CORE Generator (8). Resource utilization for Xilinx IP cores is provided by the vendor. Therefore, it is possible to estimate resource usage, to a degree, based on the operators count.

Memory. Storing 2 datasets consisting of 32,768 32-byte values will require 2 MB of memory. (Datasets that contain more than 32,768 elements can be fragmented into smaller segments and computations can be executed using these smaller segments, one at a time.) The Nallatech H101 accelerator board contains 4 SRAM banks, which are each 4 MB. Thus, we have a sufficient amount of the off-chip memory for the input data. Storing 320 8-byte elements with the computed bin counts requires ~2.5 KB of memory. The Virtex-4 LX100 FPGA contains 4,320 Kb of block RAM, thus all bin counts fit in the FPGA's BRAM.

Hardware multipliers. The Virtex-4 LX100 FPGA contains 96 XtremeDSP slices, also referred to as DSP48 by Xilinx ISE tools. Nine such slices are required to implement a double-precision floating-point multiplier (9) and our kernels only require 3 such multipliers. Thus, 27 DSP48 blocks are required, which is within the FPGA limits.

Random logic elements. The Virtex-4 LX100 FPGA contains 49,152 slices of random logic. Each double-precision floating-point multiplication operator requires 693 random logic slices in addition to the DSP48 slices (9). Likewise, each double-precision addition operator requires 821 random logic slices. Thus, implementing a dot product will require 3,721 slices of random logic. Storing 32 bin boundaries in distributed RAM made of random logic will require 256 slices, and a double-precision comparison operator requires 77 slices (10); therefore, a fully unrolled binary search tree for 32 elements will require 2,310 slices (i.e., 30 * 77). Finally, we estimate that the 32-bit integer comparison operators used in bin update loop will require 46 slices, or 460 slices in total. As a result, our total slice count is estimated to be 6,747. Of course, there are other parts of the algorithm that have not been taken into account, such as loop constructs, but estimating random logic usage beyond this point without an actual implementation becomes very difficult. We conclude, however, that our random logic utilization is within the FPGA limits.

After applying the RC amenability test, we conclude that the kernels will run 3.3-3.8 times faster on the Nallatech H101 FPGA accelerator board as compared to our microprocessor kernel implementation running on a 2.4 GHz AMD Opteron processor. We also conclude that there are sufficient resources in the Virtex-4 LX100 chip to implement the kernels.

4.2 Implementation

4.2.1 FPGA algorithm parallelization strategy

When estimating the kernel throughput in Section 3.1, we assumed that the FPGA implementation of the kernel would be fully pipelined. This requires that the two outer loops (lines 1 and 2 in the algorithms from Section 2) are merged into a single loop and the two internal loops (binary or waterfall search, line 3 in the algorithms, and bin update for the jackknife samples, lines 4 and 5 in the algorithms) are fully unrolled. (We excluded the direct bin mapping formula from the consideration for FPGA implementation due to the excessive FPGA resources required to implement *arccos* and *log* functions for double-precision numerical types.)

Unrolling bin mapping loop. The DIME-C compiler does not unroll loops automatically; therefore, we manually unroll them. When fully unrolled, the binary search requires one comparison operation less than the waterfall search, thus saving FPGA resources. It does, however, result in a more complex code since it involves nested **if-else** statements. An unrolled binary search code is also more difficult to modify when a different number of bins are needed. Nevertheless, we stick with the more conservative approach and implement an unrolled binary search algorithm for the case of 32 bins.

Unrolling bin update loop. Once the bin number is found, we must increment the bin counts for the *K*-1 jackknife samples by 1 (lines 4 and 5 in the algorithms presented in Section 2). In the reference C implementation, this is done using a **for** loop, which needs to be unrolled in order to be used inside a pipelined FPGA design. We manually unroll this loop for 10 jackknife samples. Also, since we store bin values in BRAM, updating them involves reading a value, incrementing it by one, and storing it back to BRAM. This process requires 3 clock cycles to complete, thus presenting a *read-after-write* BRAM hazard for the case when the same bin needs to be updated consecutively. In order to avoid this hazard condition, we allocate three bin arrays instead of one, and multiplex between them in the pipelined loop. The drawback to this approach, however, is we have increased BRAM usage by a factor of 3 and increased the complexity of the FPGA design.

The obvious drawback of manual loop unrolling is the loss of generality as we hard-code the number of bins and jackknife samples used in the FPGA design.

Innermost loop pipelining. Since we manually unrolled the two innermost loops, the next innermost loop (the *j* **for** loop shown in line 2 of the algorithms) is pipelined by the DIME-C compiler with the execution rate of one clock cycle per loop iteration as long as we can ensure no more than one read from each of four off-chip memory banks inside that loop.

Fusing nested loops. The execution time of the FPGA implementation of the cross-correlation algorithm in which the innermost loop is fully pipelined equals $M((N+P_N)+P_M)$ clock cycles where M is the number of iterations in the outer loop, N is the number of iterations in the inner (pipelined) loop, P_N is the pipelined loop depth, and P_M is the overhead of the outer loop. It can be reduced to $MN+(P_N+P_M)$ clock cycles when fusing the two loops and pipelining the resulting single loop. Fusing loops in the cross-correlation algorithm is straightforward. However, fusing loops in the auto-correlation algorithm is not possible within the language constructs supported by the DIME-C compiler because of the dependency of the number of loop iterations in the inner loop on the number of loop iterations in the outer loop. We decided not to fuse the loops for the cross-correlation algorithm as well, thus paying an $M(P_N+P_M)$ clock cycles penalty, but allowing us to provide a single FPGA implementation of both algorithms, analogous to the reference C implementation. The advantage of this approach is that it eliminates the need to reload FPGA bit files during the execution of the application as auto- and cross-correlation calculations are performed sequentially.

4.2.2 Data storage strategy

Ensuring innermost loop pipelining requires that there is no more than one read access to each of the off-chip SRAM banks per loop iteration. (Note that double data rates are supported on H101, but our kernel implementation is not designed to take advantage of it.) This is easy to accomplish by storing points from one dataset across all four SRAM memory banks: x coordinates are stored in one bank, y coordinates are stored in another bank, and so on. Points from another dataset are stored in the same SRAM banks, as shown in Figure 8. They are only accessed from the outer loop and therefore their reads do not interfere with the pipelining of the innermost loop.

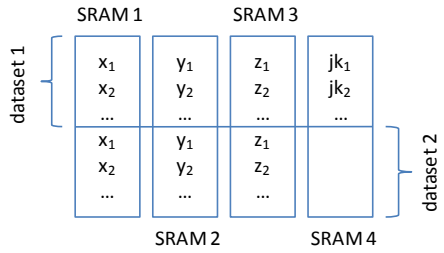

Figure 8. The SRAM input data storage model.

4.2.3 DIMEtalk network design

Implementing applications in the DIMEtalk/DIME-C environment consists of two main steps: DIMEtalk network design and DIME-C kernel implementation. The DIMEtalk network, shown in Figure 9, defines the connectivity between different modules that are used in the application. One such module is the DIME-C component that represents the actual kernel. Other modules are off-chip SRAMs, the PCI-X host interface, a clock signal generator, routers, and a memory map module. The SRAM ports in the DIME-C design do not connect directly to the SRAM, instead they interface with it through a DDR-II bridge. All these additional modules are supplied by Nallatech.

4.2.4 DIME-C implementation

The DIME-C implementation for our algorithms is shown in Figure 10. A special keyword, **sram**, is used to indicate that arrays x, y, z, and jk are stored in SRAM (in double-precision floating-point format). We declare memory to store 65,536 points per each SRAM bank, or up to 32,768 elements per dataset. The module also expects several other parameters, such as the actual number of points in each dataset, the number of bins and jackknife samples, and a flag indicating the mode of operation (auto- or cross-correlation).

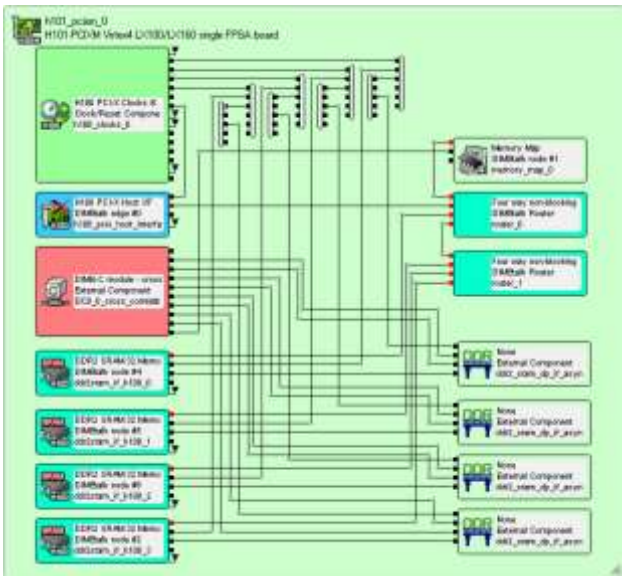

Figure 9. The DIMEtalk network design.

One of SRAMs is used to transfer computed bin counts. Compiler limitations require this SRAM declared in the same way as the rest of the memory banks, e.g., as *double*, even though the bin count results are stored internally in BRAM as 64-bit integers. The same SRAM bank is used to transfer the bin boundaries to the kernel that are then internally copied to distributed memory, and to transfer computed bin counts to the host.

The DIME-C module execution starts by copying bin boundaries from the SRAM bank referred to as BINV in the function prototype, and zeroing the BRAM used to store computed bin counts. Next, two nested loops are implemented following the FPGA algorithm parallelization strategy described previously. Finally, the bin counts stored in the three separate BRAM arrays are added together and copied to the SRAM bank for final transfer to the host system.

```
01  #define NBINS 32
02  #define NJK 10
03  #define N1 32768
04  #define N2 32768
05  #define N1pN2 65536
06
07  void cross_correlation(sram double X1[N1pN2], sram double X2[N1pN2], sram double Y1[N1pN2],
08              sram double Y2[N1pN2], sram double Z1[N1pN2], sram double Z2[N1pN2], sram double JK[N1pN2],
09              sram double BINV[N1pN2], int n1, int n2, int nb, int njk, int doSelf)
10  {
11    int i, j, k, jk, offset, indx, nb1, nb2, i_end, j_start, val;
12    double x1, y1, z1, x2, y2, z2, dot;
13    double binb00, binb01, binb02, binb03, binb04, binb05, binb06, binb07, binb08, binb09, binb10, binb11;
14    double binb12, binb13, binb14, binb15, binb16, binb17, binb18, binb19, binb20, binb21, binb22, binb23;
15    double binb24, binb25, binb26, binb27, binb28, binb29, binb30;    // NBINS
16    int binv1_00[NBINS], binv2_00[NBINS], binv3_00[NBINS], binv4_00[NBINS];  // NJK
17    int binv1_01[NBINS], binv2_01[NBINS], binv3_01[NBINS], binv4_01[NBINS];
…
26    int binv1_10[NBINS], binv2_10[NBINS], binv3_10[NBINS], binv4_10[NBINS];
27    int bin_bank, val0, val1, val2, val3, val4, val5, val6, val7, val8, val9, val10;
28
29    nb1 = nb + 1;  nb2 = nb + 2;    // make a local copy of bin boundaries
30    for (i = 0; i < nb1; i++) {  // copy bin boundaries
31        binb00 = binb01; binb01 = binb02; binb02 = binb03; binb03 = binb04; binb04 = binb05; binb05 = binb06;
…
```

```
35      binb24 = binb25; binb25 = binb26; binb26 = binb27; binb27 = binb28; binb28 = binb29; binb29 = binb30;
36      binb30 = BINV[n1 + i];
37    }
38    for (i = 0; i < nb2; i++) {  // clear BRAM for bin counts
39      binv1_00[i] = 0; binv2_00[i] = 0; binv3_00[i] = 0; binv4_00[i] = 0;
…
49      binv1_10[i] = 0; binv2_10[i] = 0; binv3_10[i] = 0; binv4_10[i] = 0;
50    }
51    if (doSelf) i_end = n1-1; else i_end = n1;
52
53    for (i = 0; i < i_end; i++) {        // loop thru one dataset
54      x1 = X1[i];  y1 = Y1[i]; z1 = Z1[i]; jk = (int)JK[i];
55      if (doSelf) j_start = i+1; else j_start = 0;
56      for (j = j_start; j < n2; j++) {         // loop thru second dataset
57        if (doSelf) offset = j; else offset = n1 + j;
58        x2 = X2[offset]; y2 = Y2[offset]; z2 = Z2[offset];
59        dot = x1 * x2 + y1 * y2 + z1 * z2;           // dot product
60        // unrolled binary search
61        if (dot < binb15) { if (dot < binb23) { if (dot < binb27) { if (dot < binb29) { if (dot < binb30) indx = 31;
62        else indx = 30; } else { if (dot < binb28) indx = 29; else indx = 28; } } else { if (dot < binb25)  {
63        if (dot < binb26) indx = 27;  else indx = 26; } else { if (dot < binb24) indx = 25; else indx = 24; } } }
64        else { if (dot < binb19) { if (dot < binb21) { if (dot < binb22) indx = 23; else indx = 22; } else {
65        if (dot < binb20) indx = 21; else indx = 20; } } else { if (dot < binb17) { if (dot < binb18) indx = 19;
66        else indx = 18; } else { if (dot < binb16) indx = 17; else indx = 16; } } } }
67        else { if (dot < binb07) { if (dot < binb11) { if (dot < binb13) { if (dot < binb14) indx = 15; else indx = 14; }
68        else { if (dot < binb12) indx = 13; else indx = 12; } } else { if (dot < binb09) { if (dot < binb10) indx = 11;
69        else indx = 10; } else { if (dot < binb08) indx = 9; else indx = 8; } } } else { if (dot < binb03) { if (dot < binb05) {
70        if (dot < binb06) indx = 7; else indx = 6; } else { if (dot < binb04) indx = 5; else indx = 4; } } else {
71        if (dot < binb01) { if (dot < binb02) indx = 3; else indx = 2; } else { if (dot < binb00) indx = 1; else indx = 0; }}}}
72
73        bin_bank = j % 4;  // update bin values
74        if (jk != 0) val0 = 1; else val0 = 0;              // update corresponding bin for jk=0
75        if (bin_bank == 0) binv1_00[indx] = binv1_00[indx] + val0;
76        else if (bin_bank == 1) binv2_00[indx] = binv2_00[indx] + val0;
77        else if (bin_bank == 2) binv3_00[indx] = binv3_00[indx] + val0;
76        else binv4_00[indx] = binv4_00[indx] + val0;
…
127     }
128   }
129
130   for (i = 0; i < nb2; i++)     {    // copy results back to SRAM
131       BINV[i] = (double)((binv1_00[i] + binv2_00[i]) + (binv3_00[i] + binv4_00[i]));
132       BINV[nb2+i] = (double)((binv1_01[i] + binv2_01[i]) + (binv3_01[i] + binv4_01[i]));
133       BINV[2*nb2+i] = (double)((binv1_02[i] + binv2_02[i]) + (binv3_02[i] + binv4_02[i]));
134       BINV[3*nb2+i] = (double)((binv1_03[i] + binv2_03[i]) + (binv3_03[i] + binv4_03[i]));
135       BINV[4*nb2+i] = (double)((binv1_04[i] + binv2_04[i]) + (binv3_04[i] + binv4_04[i]));
136       BINV[5*nb2+i] = (double)((binv1_05[i] + binv2_05[i]) + (binv3_05[i] + binv4_05[i]));
137       BINV[6*nb2+i] = (double)((binv1_06[i] + binv2_06[i]) + (binv3_06[i] + binv4_06[i]));
138       BINV[7*nb2+i] = (double)((binv1_07[i] + binv2_07[i]) + (binv3_07[i] + binv4_07[i]));
139       BINV[8*nb2+i] = (double)((binv1_08[i] + binv2_08[i]) + (binv3_08[i] + binv4_08[i]));
140       BINV[9*nb2+i] = (double)((binv1_09[i] + binv2_09[i]) + (binv3_09[i] + binv4_09[i]));
141       BINV[10*nb2+i] = (double)((binv1_10[i] + binv2_10[i]) + (binv3_10[i] + binv4_10[i]));
142   }
143 }
```

Figure 10. The DIME-C kernel implementation.

4.2.5 Compilation

Nallatech's DIME-C Function Generator translates DIME-C code to VHDL. It also reports slice resource

utilization of 11,118 slices, which is about ¼ of the slices available on the LX100 FPGA. The DIMEtalk network design shown in Figure 9 itself requires on the order of 10,000 slices. Thus, we estimate that our overall design will occupy ~21,000 slices, which is close to half of the slices available in the LX100 FPGA (but is about 3 times as much as was in our initial estimate in Section 4.1.3). Therefore, we consider one more implementation in which we partially unroll the innermost loop to perform two calculations per iteration. In this case, the DIME-C Function Generator reports slice utilization at 17,563 slices.

Once DIME-C is translated to VHDL, the DIMEtalk System Design tool is used to generate VHDL code and user constraints files for the entire DIMEtalk network design, and the Xilinx ISE is used to synthesize, place, and route the design for the FPGA. Table 4 presents the final resource utilization for these two designs, which both meet timing requirements for 100 MHz operational frequency.

	Design 1	Design 2	Total available
DSP48s	36	63	96
RAMB16s	79	112	240
Slices	23,893	33,951	49,152

Table 4. LX100 resources utilization reported by PAR.

4.2.6 Host interface

The host interface with the FPGA kernel is straightforward. We use the DIMEtalk API to initialize the FPGA, configure it with the bit file, send data, start the execution of the design, poll the status of the FPGA to find out when it is done, and transfer the results back to the host system. The input data is rearranged on the host system into 4 arrays. The DIMETalk_Write subroutine is called 5 times: first to send an array containing input parameters such as the number of data points in each dataset, and next to transfer the four arrays that contain the actual input datasets. Finally, the DIMETalk_Read subroutine is called to transfer the computed bin counts from SRAM to the host.

4.3 Results and discussion

We first consider a dataset of 32,768 observed objects and 32,768 random objects, and calculate the TPACF using 5 bins per decade of scale for angular separations from 0.01 to 10,000 arcminutes for 10 jackknife samples and the full sample itself. We measure data transfer times (Table 5) and design execution time (Table 6) for the two implementations.

	Alg1	**Alg2**
Data in (bytes)	1,048,824	1,835,266
Data in (sec)	0.0048	0.0067
Data out (bytes)	2,816	2,816
Data out (sec)	0.000057	0.000046

Table 5. Data transfer time.

We compare data transfer times shown in Table 5 with the predicted data transfer times shown in Table 3. We observe that the actual transfer time is significantly shorter than predicted. Our predictions were based on micro-benchmarks for Nallatech H101 accelerator reported in (7). Our own micro-benchmarks that match our application characteristics suggest that the fraction of ideal throughput performing useful communication is much higher: α_{write}=0.219 and α_{read}=0.049. Thus, the lesson here is not to rely on micro-benchmarks for probing hardware characteristics performed by others since they do not reflect our application characteristics.

The measurements shown in Table 6 "FPGA design 1" column are directly comparable to the predicted $t_{RC_{SB}}$ and *speedup* values from Table 3. Both measured auto-correlation (DD in Table 6 and **Alg1** in Table 3) and cross-correlation (DR in Table 6 and **Alg2** in Table 3) values are within 1% of their estimated values. The RR values in Table 6 correspond to the auto-correlation algorithm (**Alg1**) in which no jackknife re-sampling was used.

kernel	CPU	FPGA design 1		FPGA design 2	
		time	speedup	time	speedup
DD (sec)	17.8	5.436	3.3×	2.718	6.6×
RR (sec)	10.8	5.436	2.0×	2.718	4.0×
DR (sec)	40.5	10.816	3.8×	5.408	7.5×

Table 6. Kernel execution time for 32K dataset.

Table 7 presents the execution time for a dataset consisting of 97,178 objects. In this case, the FPGA kernel is invoked multiple times with segments of data that do not exceed 32,768 points. Overall application speedup achieved with our second FPGA design is 6.7x as compared to the microprocessor implementation.

kernel	CPU	FPGA design 1		FPGA design 2	
		Time	speedup	time	speedup
DD, *sec*	153.5	47.3	3.3×	23.8	6.5×
RR, *sec*	103.3	47.3	2.2×	23.8	4.3×
DR, *sec*	375.8	94.6	4×	47.5	7.9×
total, *sec*	632.6	189.2	3.3×	95.1	**6.7×**

Table 7. Kernel execution time for 97K dataset.

To conclude, in spite of the incorrectly predicted data transfer time, the actual performance of the kernels implemented on the Nallatech H101 FPGA accelerator board matches their predicted performance. This, of course, would not have been the case if the algorithms were I/O-bound instead of compute-bound. However, our FPGA resources utilization predictions differ significantly from the actual utilization. Based on the algorithm description, we estimated slices utilization at ~7,000. After implementing the algorithm in DIME-C, we updated our predictions to ~11,000 slices. After hardware synthesis we found out that the design occupies ~24,000 slices, which is about 3.5 times more than we predicted.

5 GPU implementation

Our GPU implementation of TPACF is designed to run on NVIDIA GPUs. The basic processing unit in NVIDIA GPUs, starting from G80 architecture, is the streaming processor (SP), a fully pipelined, single-issue, in-order microprocessor complete with two ALUs and an FPU. A group of 8 SPs, 2 additional special function units (SFUs) for transcendental operations, and 16kB of shared memory form a streaming multiprocessor (SM). Each SM has a small instruction cache and a read only data cache. A group of 2 or 3 SMs with some additional memory form texture/processor cluster (TPC). Several such clusters form a streaming processor array (SPA). As an example, GPUs used in GeForce GTX 280 (11) have 240 SPs that run at 1.3 GHz (Figure 11). A 512-bit interface to GDDR3 memory provides 141.7 GB/s bandwidth. Programming for NVIDIA GPUs is done in CUDA programming language using CUDA SDK (12).

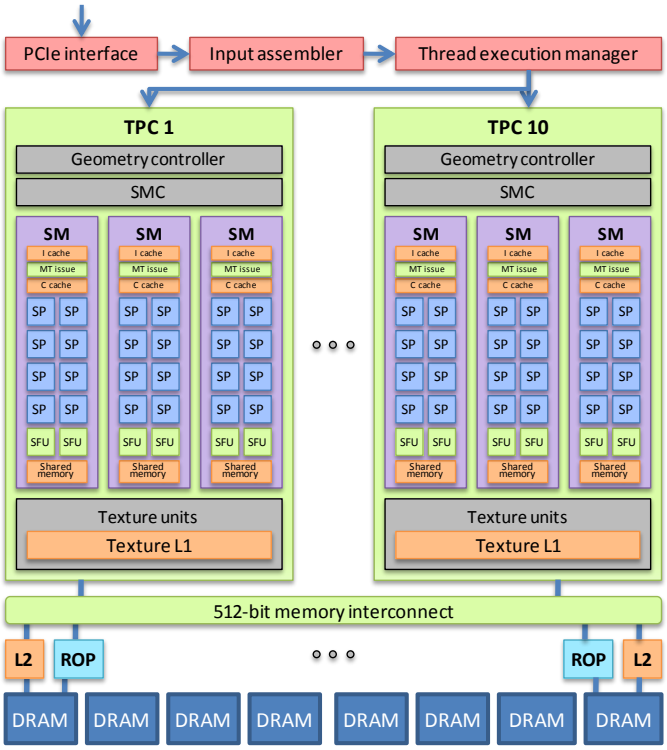

Figure 11. NVIDIA GeForce GTX 280 GPU architecture.

5.1 Suitability of GPUs for TPACF

Computing the angular separation histograms $DD(\theta)$, $DR(\theta)$, and $RR(\theta)$ for N_D~100,000 points and n_R~100 random datasets is time-consuming. As a specific example, it takes over 13 hours to run this code for N_D=97178 and n_R=100 on a single core of a 2.4 GHz AMD Opteron processor. Yet the problem is highly parallelizable with substantial data reuse opportunities. In fact, each point is used N_D times during each invocation of the `doHistogram` kernel, and, furthermore, each pair of points can be treated independently, allowing $O(N_D^2)$ parallel calculations. Assuming that a group of N_t~100 points can be pre-loaded and used N_t times ($N_t << N_D$) in the dot product calculation, 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. As, assuming double-precision, this rate is comparable to the peak rate of ~1.8 bytes per flop on NVIDIA GeForce GTX280GPU, we can potentially achieve a substantial portion of the peak GPU performance, at least for the dot product part of the kernel.

However, not all aspects of the TPACF lend themselves so easily to efficient GPU implementation. Most significant among the difficulties inherent in a GPU implementation of the TPACF is the creation of the histograms, especially in lieu of the necessity of jackknife re-sampling. Due to the low performance of atomic memory operations on the NVIDIA G200 architecture (and their non-existence on the G80 architecture), the limited performance of scattered writes to global memory, and the limited amount of shared memory available, an efficient histogram updating algorithm supporting an arbitrary number of bins is infeasible. However, by restricting the number of bins to be less than 64, which is a reasonable

number for most astrophysical application, we can leverage the 64 bin GPU histogram algorithm described in (13), which achieves good performance.

The other significant difficulty in computing angular separation histograms on the GPU is due to the limited numerical precision of the NVIDIA G80 architecture. This has implications on two aspects of the computation. First, the single-precision (32-bit) floating-point arithmetic supported on the G80 architecture is not sufficient to represent angular separations below 1 arcminute, whereas in practice we frequently need to perform the calculations down to 0.01 arcminute. Second, the unavailability of 64-bit integers on the G80 architecture complicates the bin accumulation as bin counts quickly overflow 32-bit storage, thereby requiring special code to prevent this overflow condition. Note that these two issues are not present on the NVIDIA G200 architecture on which double-precision (64-bit) floating-point arithmetic is supported. However, since the peak double-precision floating-point performance on the G200 architecture is only about $1/12^{th}$ of the peak single-precision performance, it still may be advantageous to use single-precision floating-point arithmetic even on G200 architecture.

5.2 TPACF CUDA implementation

Our GPU implementation of the original C subroutine shown in Figure 5 consists of two separate kernels. One kernel, corresponding to lines 5-12 of the code shown in Figure 5, takes as input two sets of points and writes histogram bin assignments for every pair of points to the GPU global memory. The second kernel, corresponding to line 13 of code shown in Figure 5, constructs a histogram from the output of the first kernel. These two kernels are repeatedly called in sequence on subsets of the data points and random points to compute sub-histograms, which are combined appropriately by the CPU into complete $DD(\theta)$, $DR(\theta)$, and $RR(\theta)$ histograms.

To avoid overflowing the 32-bit integer storage used for bin counts, we divide the samples of points into smaller subsets such that the product of the first input size and the second input size is less than 2^{32}-1, thus overcoming the lack of 64-bit integer support on the G80 GPU architecture. In particular, in this implementation we partition the datasets into subsets consisting of no more than 16,384 points. Since each histogram bin assignment written to the GPU global memory by the first kernel requires 1 byte of memory, the total amount of memory required to store the output of the first kernel is rather large and an adjustment may need to be done to the data partitioning size to ensure that there is sufficient memory on the GPU card. It may be necessary on cards with less memory to reduce the input size, and it may be advantageous on cards with more memory to increase the input size, keeping in mind the 32 bit integer limitation.

The principal reason for dividing the work between two GPU kernels rather than implementing one combined kernel is due to the jackknife re-sampling algorithm, which will be discussed later. In short, the combined kernel would need to support variable dataset sizes, which is easy enough to implement, but would prevent the compiler from unrolling the main loop, thereby causing a degraded performance. Dividing work between two kernels has other advantages as well. We can take advantage of workload balancing schemas appropriate for each individual kernel: the bin assignment kernel runs on a 2D grid of thread blocks, whereas the histogram kernel runs on a 1D grid of thread blocks.

5.2.1 Bin assignment kernel

The first kernel computes and outputs bin assignments for every pair of points in the two input data sets. It can be visualized as producing a 2D grid of values in which each element of the grid corresponds to one pair of points, as shown in Figure 12. The computations on the grid are divided between blocks of threads where each thread block corresponds to a specific sub-grid. The sub-grids are square to

simplify taking advantage of the symmetry in the *DD(θ)* and *RR(θ)* computations. For an N×N sub-grid (N=128 in our implementation) a block will have N threads. Each thread serially computes the histogram bin assignment for N pairs and writes them to memory, packing four of them into a single 32 bit integer. For this reason it is necessary that N be a multiple of 4. This arrangement allows all reads and writes involving global memory to be coalesced for maximum performance.

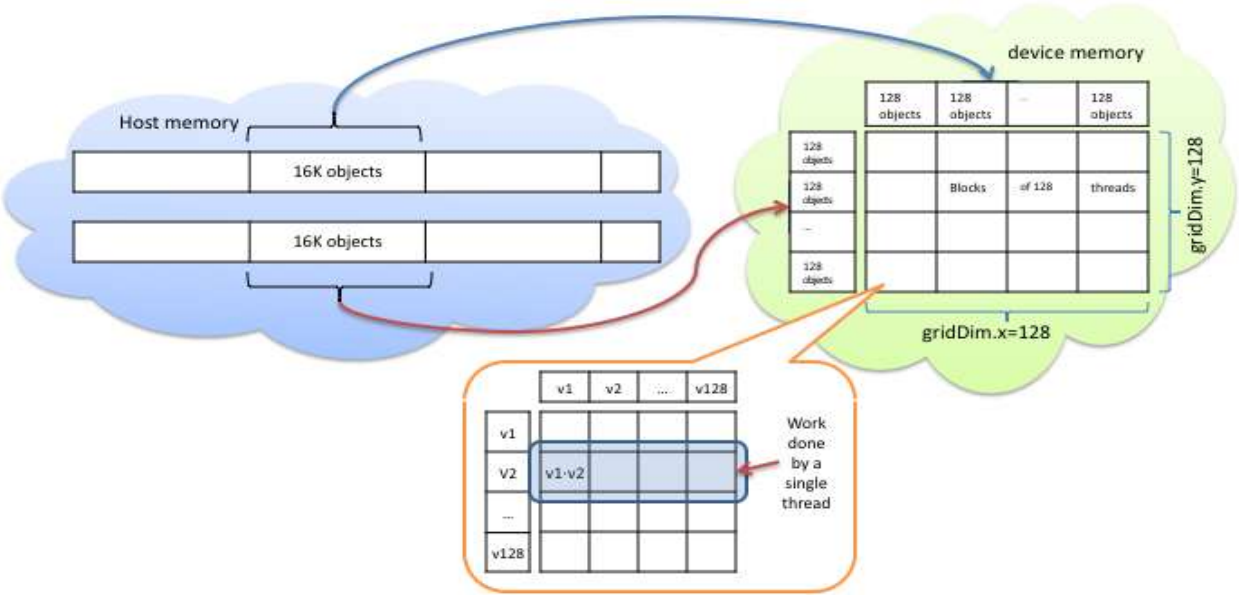

Figure 12. The mapping between data in the host memory, the data in the device memory, and the work assignment between blocks of threads and threads in each block.

The inputs to this kernel are given as two arrays of structs, which contain 3 floating-point values and 1 integer value in case of the single-precision implementation and as two structs of pointers to arrays containing floating-point values in case of double-precision implementation. The integer is used to store the jackknife assignment, although it is unused by this kernel. NVIDIA GPUs are capable of coalesced memory reads only with strides of 4, 8, or 16 bytes, we must, therefore, make extra provisions to ensure proper memory alignment. In particular, different memory data arrangements are used for single-precision and double-precision floating-point versions of the kernel. Figure 13 presents the double-precision floating-point implementation of the kernel used to calculate *DR(θ)* histogram bin assignments. The kernel for computing *DD(θ)* and *RR(θ)* bin assignments is similar, but with additional provisions made for taking advantage of symmetry, as described below.

Once a dot product is computed to determine an angle (line 22 in Figure 13), a waterfall search is used to determine the appropriate bin assignment. This operation is implemented as a manually unrolled sequential search (lines 23-54 in Figure 13). This is obviously a performance bottleneck due to the branch divergence. However, binary search and explicit bin computation methods described earlier both proved to offer significantly worse performance given the nature of this calculation.

Symmetry can be taken advantage of when computing *DD(θ)* and *RR(θ)*. Doing so requires that one bin out of the maximum 64 be reserved for elements that do not need to be computed. When taking advantage of symmetry, each block determines whether it is above, below, or on the main diagonal of

the grid. If it is above, it simply writes the reserved bin to each grid element for which it is responsible. If it is below, it computes the bin assignments normally. If it is on the main diagonal, it is necessary to add a check to determine whether or not a given pair should be computed. This adds another source of branch divergence, but the performance gains offered by taking advantage of symmetry far outweigh this pitfall.

```
01 __global__ void binKernel(cartesian g_idata1, cartesian g_idata2, unsigned int *g_odata)
02 {
03    extern __shared__ double3 sdata[];                  // shared memory used to store vectors from g_idata2
04    double dotp;  unsigned int indx4;  double3 vec1, vec2;        // temporary variables
05    int tx = (blockIdx.x<<7) + threadIdx.x;                     // tx is the "x position" in the grid
06    int by = (blockIdx.y<<7);                                   // "y position" depends on i, this is just y block
07    vec1.x = g_idata2.x[tx];          // each thread reads one vector from global to a register, its "assigned vector"
08    vec1.y = g_idata2.y[tx];    // reads are coalesced, as cartesians are aligned properly and there are no conflicts
09    vec1.z = g_idata2.z[tx];
10    sdata[threadIdx.x].x = g_idata1.x[by+threadIdx.x];   // read one unique vector from global to shared per thread,
11    sdata[threadIdx.x].y = g_idata1.y[by+threadIdx.x];   // the "shared vectors"
12    sdata[threadIdx.x].z = g_idata1.z[by+threadIdx.x];   // reads are coalesced for the same reason
13    __syncthreads();                      // ensure all reads are finished before using them for any calculations
14    by <<= (LOG2_GRID_SIZE - 2);  by += tx;          // compute address for packed data storage, needed later
15    #pragma unroll
16    for (int i=0; i<128; i+=4) {   // Iterate through 128 vectors in sdata
17       indx4 = 0;
18       #pragma unroll
19       for (int j=0; j<4; j++) {   // 4 vectors per 1 int output
20          vec2 = sdata[i+j];          // sdata broadcasts sdata[i+j] to all threads in a block to avoidbank conflicts
21          // each thread computes the dot product of its assigned vector with every shared vector
22          dotp = vec1.x*vec2.x + vec1.y*vec2.y + vec1.z*vec2.z;
23          if (dotp < binbounds[30]) indx4 += (124<<(j<<3));    // the jth byte of indx4 is always a multiple of four;
24          else if (dotp < binbounds[29]) indx4 += (120<<(j<<3));  // because of the histogram kernel,
25          else if (dotp < binbounds[28]) indx4 += (116<<(j<<3));  //  this schema supports at most 64 bins,
26          else if (dotp < binbounds[27]) indx4 += (112<<(j<<3));  // and thus ignores the two least significant bits
… // sequential search loop is manually unrolled
52          else if (dotp < binbounds[1]) indx4 += (8<<(j<<3));
53          else if (dotp < binbounds[0]) indx4 += (4<<(j<<3));
54          else indx4 += (0<<(j<<3));
55       }
56       g_odata[by+(i<<(LOG2_GRID_SIZE - 2))] = indx4;          // pack 4 bin indices into an unsigned int
57    }
58 }
```

Figure 13. The source code of the double-precision kernel that computes bin assignments and stores them in global memory.

5.2.2 Histogram kernel

The second kernel, which constructs a histogram from the bin assignments written to the global memory by the first kernel, is based on the NVIDIA whitepaper on GPU histogramming (13). The source code of the kernel is shown in Figure 14. It employs a predefined number of threads per block and the necessary number of blocks to create a number of per-block sub-histograms in global memory, which are then compiled into a single histogram in the host memory. The per-block sub-histograms are created by first having each thread create a per-thread sub-histogram of 252 elements (lines 10-16 in Figure 14). The number 252 is chosen to avoid overflow in the byte counters used to store the per-thread sub-histograms. The per-thread sub-histograms are stored in shared memory, and a method

described in the NVIDIA whitepaper is used to ensure that no bank conflicts occur in the shared memory. After this, a subset of each block's threads, equal to the number of bins, compiles these per-thread sub-histograms into the per-block sub-histogram (lines 18-28 in Figure 14). Next, a small helper kernel, shown in Figure 15, compiles the per-block sub-histograms into a smaller number of sub-histograms. Compiling the small number of remaining sub-histograms into a full histogram is left to the CPU. This helper kernel can be eliminated on GPUs with compute capability 1.1 or greater by using atomic memory operations, but doing so results in some loss of performance.

```
01 __global__ void histoKernel(unsigned int* g_odata, unsigned int* g_idata, int size)
02 {
03    const int threadPos = (threadIdx.x & (~63)) | ((threadIdx.x & 15) << 2) | ((threadIdx.x & 48) >> 4);
04    __shared__ unsigned char s_Hist[MEMPERBLOCK];                // stores all per-thread sub-histograms
05    for (int pos = threadIdx.x; pos < (MEMPERBLOCK >> 2); pos += blockDim.x) ((unsigned int *)s_Hist)[pos] = 0;
06    __syncthreads();
07    const int gStart=__mul24(blockIdx.x,DATAPERBLOCK);// location in g_idata in which this block starts reading
08    const int blockData = min(size - gStart, DATAPERBLOCK);    // amount of data to be processed by this block
09    unsigned int dataTemp;
10    for (int pos = threadIdx.x; pos < blockData; pos += blockDim.x) {
11       dataTemp = g_idata[gStart + pos];                        // read integer from global memory
12       s_Hist[threadPos + __mul24( (dataTemp >>  2) & 63, NUMTHREADS)]++;      // increment appropriate
13       s_Hist[threadPos + __mul24( (dataTemp >> 10) & 63, NUMTHREADS)]++;      // bins in shared memory
14       s_Hist[threadPos + __mul24( (dataTemp >> 18) & 63, NUMTHREADS)]++;
15       s_Hist[threadPos + __mul24( (dataTemp >> 26) & 63, NUMTHREADS)]++;
16    }
17    __syncthreads();
18    if (threadIdx.x < NUMBINS) {                 // use NUMBINS threads to create a per-block sub-histogram
19       unsigned int sum = 0;                    // from the data in shared memory
20       const int tid = threadIdx.x;             // each thread calculates the total number of elements in bin tid
21       const int hStart = __mul24(tid, NUMTHREADS);          // starting point in the histogram
22       const int accumStart = (threadIdx.x & 15) * 4;        // another trick to ensure no bank conflicts
23       for (int i=0, accum=accumStart; i<NUMTHREADS; i++) {  // Iterate through thread sub-histograms' tid bins
24          sum += s_Hist[hStart + accum];                     // to calculate the sum
25          if (++accum == NUMTHREADS) accum = 0;
26       }
27       g_odata[blockIdx.x * NUMBINS + tid] = sum;            // write to global memory
28    }
29 }
```

Figure 14. The histogram kernel based on an NVIDIA white paper **(13)**.

```
01 __global__ void mergeKernel(unsigned int* d_iodata, int numBlocks)
02 {                              // merges numBlocks per-block sub-histograms into gridDim.x sub-histograms
03    const int size = numBlocks * NUMBINS;                  // total number of histogram bins
04    const int gPos = blockIdx.x * NUMBINS + threadIdx.x;   // (starting) position in global memory for this thread
05    const int numThreads = gridDim.x * blockDim.x;         // number of threads
06    unsigned int sum = 0;                                  // compute bin counts for new sub-histograms
07    for (int pos = gPos; pos < size; pos += numThreads)  sum += d_iodata[pos];
08    d_iodata[gPos] = sum;             // write to memory, overwriting the (now useless) first portion of d_iodata
09 }
```

Figure 15. The helper kernel used to add up the per-block sub-histograms into a smaller number of sub-histograms.

5.2.3 Jackknife Re-sampling

Handling jackknife re-sampling efficiently for a general case turned out to be the most difficult task in this implementation. We discovered, however, that by imposing reasonable restrictions on the number of jackknife samples, we could achieve jackknife re-sampling essentially for free. In the CPU implementation, jackknife re-sampling is handled by having `njk+1` histograms, where `njk` is the number of jackknives. For each angle, the appropriate bin is determined and is incremented in `njk` of these bins, ignoring the bin from which this particular angle is removed. On the other hand, the GPU implementation has `njk` histograms, each representing the inverse of the corresponding jackknife. The i^{th} inverse jackknife is defined as the histogram of those elements that are removed from the i^{th} jackknife. Denoting these inverse jackknives as I_1, I_2, ..., I_{njk}, it is possible to reconstruct the histogram of all data by summing the inverse jackknives, as each element is accounted for in precisely one inverse jackknife. The i^{th} jackknife can be reconstructed by subtracting I_i from the histogram of all data. By presorting the data according to jackknife, the GPU histogram kernel can be invoked on large sets of data which correspond solely to one inverse jackknife, resulting in very little loss in efficiency versus invoking the histogram kernel on the entire output of a call to the first kernel. Computing the histogram of all data and the jackknives is very little work, and thus is relegated to the CPU. (Note this same idea applies to the CPU implementation as well, but has not yet been implemented in our reference C implementation.)

5.2.4 Single/double-precision considerations

Cartesian coordinates are stored in the host and device memory as a single array of `struct { float x,y,z; int jk; }` elements aligned at 16-bytes boundary in the single-precision kernel, and as three separate arrays, `x`, `y`, `z`, in the double-precision memory. This is to ensure optimal memory bandwidth. Few adjustments are made in the GPU kernels to reflect this difference, but otherwise, no additional modifications to the code are necessary.

5.2.5 Multi-GPU implementation

There are other portions of TPACF which could be parallelized, such as the computation of *DR(θ)* and *RR(θ)* histograms for n_R datasets with random data. This parallelism can be easily taken advantage of on a multi-GPU system by spreading the calculations between multiple GPUs in a single system or between multiple compute nodes with GPUs attached to them, such as NCSA's QP cluster (14). We implemented an MPI-based version of the application that runs on a 16-node cluster, where each node has four CPU cores and four GPUs. Since there is a 1:1 CPU core to GPU ratio on each cluster node, running the MPI application is straightforward as each MPI process can have an exclusive access to a single CPU core and a single GPU. MPI process ID modulo the number of GPUs per node identifies a unique GPU for each MPI process running on a given node. This requires, however, that the MPI threads are placed sequentially on each compute node rather than in round robin fashion.

5.3 Results and discussion

We first analyze the overall performance improvements of the GPU-based single-precision and double-precision floating-point implementations versus the double-precision single-core CPU implementation. The CPU code is compiled with the ICC compiler and `-fast` optimization level, while the GPU code is compiled with the NVCC compiler and `-arch_sm_10` (for G80 architecture) or `-arch_sm_13` (for G200 architecture), using CUDA SDK 2.1. The application is executed using 100 random data files, with 5 bins per decade of angular scale, a minimum angular separation of 1 arcminute, a maximum angular separation of 10,000 arcminutes, and number of data points ranging from 1024 to 97K, as shown in

Figure 16. We measure the total time needed to compute *DD(θ)*, *DR(θ)*, and *RR(θ)* histograms since this part of the code is responsible for over 99% of the overall execution time for sufficiently large data sets.

The execution time of the GPU implementation remains near-constant for small datasets up to 16,384 points since the code is designed to process 16,384×16,384 points at a time. Only when the dataset size reaches 16,384 elements do we observe a substantial performance improvement in comparison to the CPU implementation. This is the minimal data set size in our GPU implementation that does not require data padding, thus we do not perform any unnecessary data transfers and computations. At smaller dataset sizes, overall performance suffers, in part due to the overheads of data transfers and due to the unnecessary computations done on the padded data. With larger dataset sizes, we have the additional benefit of reusing segments of data multiple times without the need to transfer them multiple times to the GPU memory.

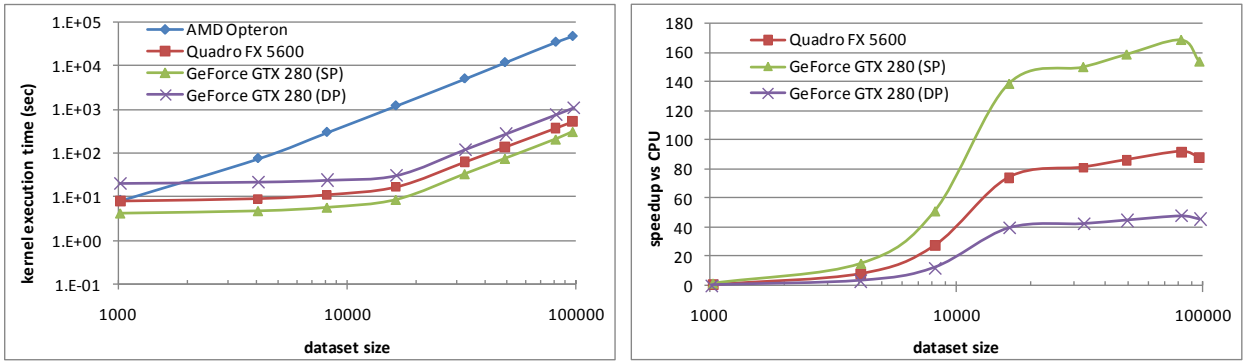

Figure 16. A comparison of the execution time (left) and speedup (right) between the AMD Opteron, Quadro FX 5600, and GeForce GTX 280 platforms.

GPU speedup drops somewhat for a dataset consisting of 97K points. This is because the size of the dataset is not a multiple of 16,384, and we thus suffer from issues similar to datasets smaller than 16,384 points due to the data padding. The vast majority of the time is spent in the bin assignment kernel (Figure 17), since this is where floating point operations are performed. The absolute time spent in the histogram kernel is identical for the single- and double-precision versions since this particular kernel operates only on integer values.

The single-precision bin assignment kernel is about 3 times faster than the double-precision version of the kernel. Overall, single-precision performance of the two combined kernels on the GeForce GTX 280 GPU is about 3.5x the double-precision performance even though the peak FLOPS ratio between the single-precision and double-precision performance for this chip is about 12. This indicates that the main performance limiting factor is not due to the floating-point performance, but due to the limited memory bandwidth. As we discussed earlier, ~1.6 bytes per flop are required by the bin assignment kernel in the double-precision version and ~0.8 bytes per flop in the single-precision version. The theoretical peak rate of the GeForce GTX 280 is ~1.8 bytes per flop for double-precision and only ~0.15 bytes per flop for single-precision, which is only about 1/5 of what is required by the bin assignment kernel. This mismatch between the application requirements and the hardware capabilities is responsible for the performance loss for the single-precession kernel.

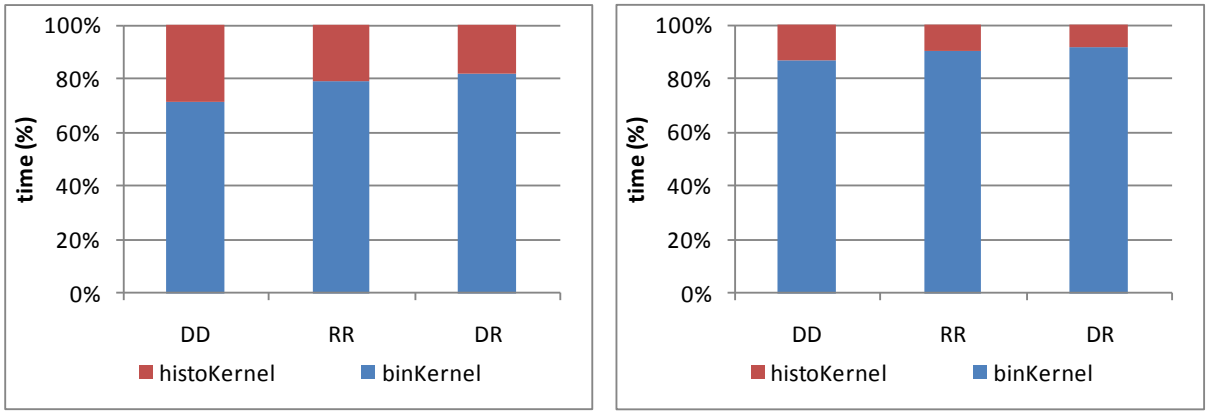

Figure 17. The time distribution between the bin mapping and histogram kernels for the dataset consisting of 3*16,384 elements for single- (left) and double- (right) precision cases running on the GeForce GTX 280.

Our MPI implementation scales linearly, as shown in Figure 18. In this case we measure total application execution time, including the GPU kernel time, file I/O, and MPI overheads. We notice that the execution time starts to level off when using larger number of compute nodes. This is because the file I/O and MPI overheads become more pronounced as the time to do the actual computations reduces proportional to the number of nodes.

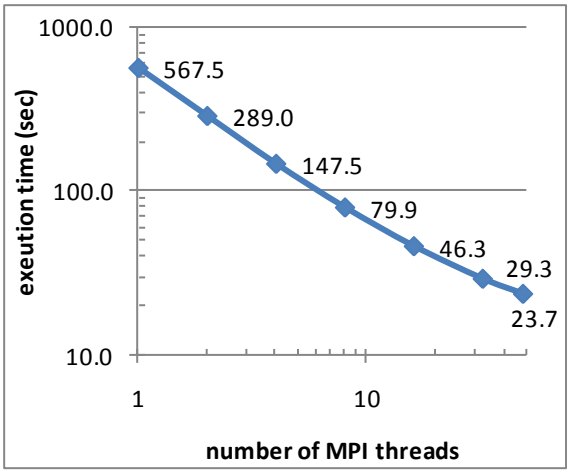

Figure 18. Multi-GPU scaling.

Finally, we analyze the accuracy of the computed results. Figure 19 contains 3 plots of the two-point angular correlation function computed for a subset of the photometrically classified quasars and random catalogs first analyzed by (2). In this case, θ runs from 0.01 to 10,000 arcminutes with 5 bins per decade of scale and 10 jackknife samples are used to estimate errors for each value of $\omega(\theta)$, shown as vertical bars centered at each point on the plot. Blue circles correspond to the results obtained on the GeForce GTX 280 GPU using double-precision, red squares correspond to the results obtained on the GeForce GTX 280 GPU using single-precision, and green triangles correspond to the results obtained on the CPU using double-precision. Note that the double-precision results for the GPU and CPU implementations coincide, whereas the single-precision GPU results are slightly off for angular separations below approximately 4 arcminutes.

Since only 41-bit fixed point arithmetic is required to achieve the 0.01 arcminute resolution needed to analyze many datasets, the numerical precision of single-precision floating-point arithmetic is not sufficient to perform the distance calculations for angular separations below approximately 1 arcminute. Therefore one obvious limitation of the single-precision GPU implementation of the kernels is the restriction on the lowest angular separation that it can compute. This limitation is fully removed in the double-precision GPU kernel implementation.

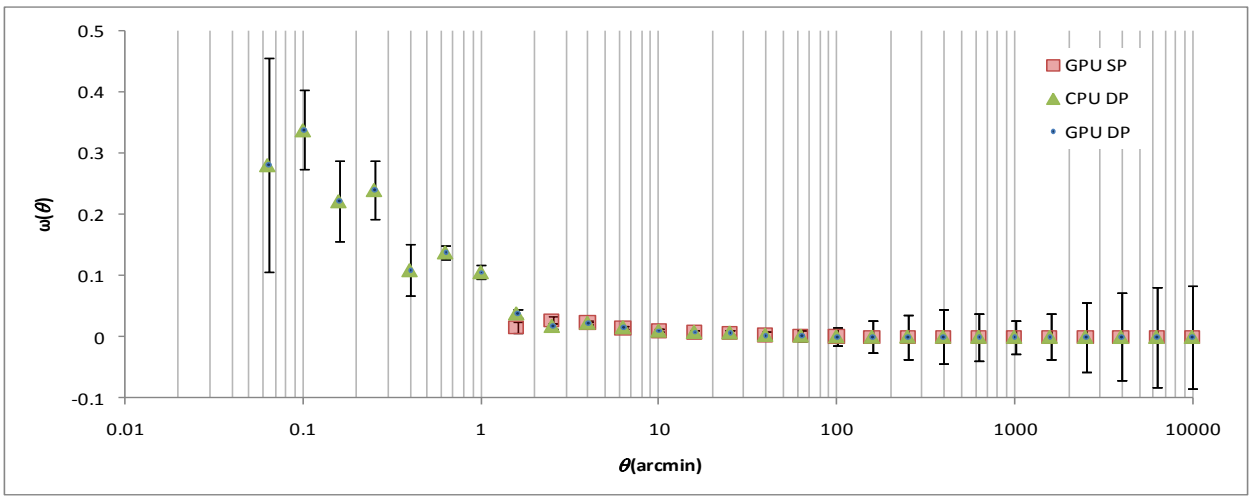

Figure 19. Two-point angular correlation computed separately using CPU and GPU.

6 Cell/B.E. implementation

In this study, a 3.2 GHz dual-Cell blade QS20 (15) server is used for single precision and QS22 (16) server is used for double precision runs. QS20 contains two Cell/B.E. (17) and QS22 contains two PowerXCell 8i (18) processors with enhanced double-precision support. The QS20 system runs Fedora Core 7 Linux OS with kernel 2.6.22-BSC and the IBM SDK for Multicore Acceleration ver. 3.0. The QS22 system runs Fedora Core 9 Linux OS with kernel 2.6.26-bsc and IBM SDK ver. 3.1 (19).

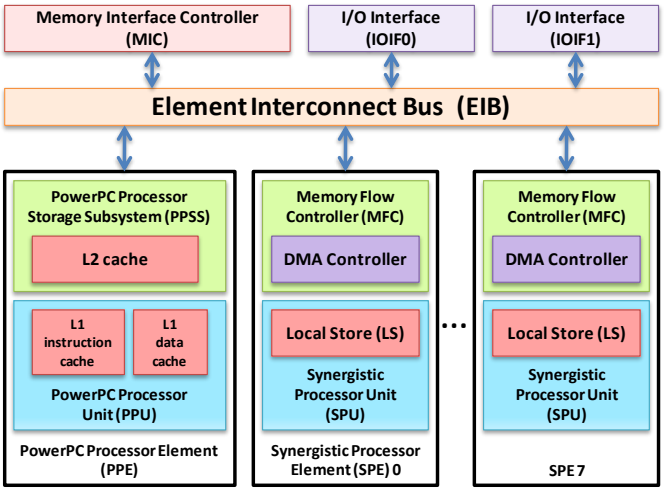

Figure 20. Cell/B.E. processor architecture.

The Cell/B.E. is a heterogeneous system consisting of one 64-bit PowerPC core called the Power Processor Element (PPE), eight Synergistic Processor Elements (SPEs), system memory, and an I/O controller (Figure 20). The processing elements are linked by an internal high-speed bus called the Element Interconnect Bus (EIB). The PPE is a 64-bit Power-Architecture-compliant core with 32KB first-level instruction and data caches and a 512-KB second-level cache. Each SPE consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC), which includes a DMA controller, a Memory Management Unit (MMU), a bus interface, and an atomic unit for synchronization with other SPEs and PPE. SPU is a Single Instruction, Multiple Data (SIMD) processor whose load and store instructions are performed in local address space.

The SPU has two execution pipelines: The floating-point and fixed-point units are on the even pipeline while the rest of the functional units are on the odd pipeline. The SPU can issue and complete up to two instructions per cycle, one on each execution pipeline. Simple fixed-point operations take two cycles,

and single-precision floating-point and load instructions take six cycles. Two-way SIMD double-precision floating-point is also supported, but the maximum issue rate is one SIMD instruction per seven cycles.

For the 3.2 GHz Cell/B.E., the EIB is capable of providing peak bandwidth of 204.8 GB/s. The memory interface controller provides 25.6 GB/s to system memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound. The eight SPUs of the Cell/B.E. processor have combined theoretical peak performance of 204.8 GFLOPS in single precision and 14.63 GFLOPS in double precision. With PowerXCell 8i, the double precision performance is improved to over 100 GFLOPs, the half of the single precision performance.

6.1 Suitability of Cell/B.E. for TPACF

The computation of angular separation histograms on Cell/B.E. can be evenly distributed across 8 SPEs, thus, potentially resulting in 8× performance improvement. In addition, when using single-precision floating-point arithmetic, each SPE operates on a vector of 4 values, thus potentially providing additional 4× performance improvement, or 32× in total. This formula is not as straightforward with the double-precision arithmetic on Cell/B.E. since SPEs can operate on just 2 64-bit values at a time and they can only issue one instruction every 7 clock cycles. This puts the potential performance improvement at 8*2*1/7≈2.3×. On the other hand, SPEs in the PowerXCell 8i processor issue a double-precision floating-point instruction every clock cycle, thus putting a potential performance improvement at 8*2=16×. Note that in this discussion we have only relied upon the raw processor speed and completely omitted memory considerations. However, since the computational complexity of the underlying algorithm is $O(N^2)$, data transfer time should be small compared to the compute time. Thus, we conclude that there are potentially substantial benefits to be gained when the algorithm is ported to Cell architecture.

However, there are several challenges in implementing TPACF on the Cell processor. To get a good performance, the code must be amenable for SIMD style of execution. While it is straightforward to shuffle four points in separate vectors for *x*, *y*, *z* coordinates and compute the dot product in SIMD fashion, it is generally not suitable to implement the search algorithm because different slots in a vector could diverge in branch instructions, making it difficult to do a direct translation from the CPU code to an efficient SIMD code in SPUs. We could, however, implement a SIMD amenable restricted comparison tree in which only the last few bin boundary values are used since most angular separations fall within the last few bins. Figure 4 demonstrates why this approach will work. This way we can eliminate most of the branch divergences at a cost of few extra calculations.

The second challenge comes from the bin counts update. The bin assignment step requires updating the bin values according to the computed indices. Although we could obtain a vector of pointers from the vector of indices, SPU does not support vector loading using a vector of pointers. Furthermore, different slots in the indices vector could contain the same index, which makes it data dependent between different slots. This issue can be addressed using a lookup table approach, as discussed later.

6.2 TPACF Cell/B.E. implementation

A single call to the compute kernel is replaced with a call to a compute kernel executed on an SPE. Each of DD, DR or RR computations is scheduled to run independently on an SPE. We accumulate intermediate results independently for each SPE and merge them when all jobs are finished. Since the problem is of $O(N^2)$ complexity, the time needed to send data to the SPEs and receive computed results back is negligible, thus, eliminating the need for double-buffering.

6.2.1 Data structures

The original data in the reference implementation is stored in an array of point structures, where *x*, *y*, *z*, and *jk* are elements within the structure. Each point structure consists of 16 bytes, thus it can be casted to a vector type in a straightforward manner. Converting this data layout to vectors, each consisting of only *x*, *y*, and *z* values for an efficient SIMD execution on the SPE requires few shuffle operations. Since each point is processed *N* times, overhead of multiple shuffling operations adds up quickly. Therefore, we modified the reference implementation to store data as an array of 4-value vectors in addition to the original array of structures storage model.

6.2.2 Bin assignment kernel

The Cell/B.E. SPE code that computes the dot product for points from two datasets is shown in Figure 21. The inner loop is incremented by 4 and the dot products are computed in SIMD manner resulting in 4 results at a time. Note that the data accessed from the inner loop is structured as described above whereas data accessed from the outer loop is stored in the same way as in the reference implementation since in the outer loop we only access one point at a time. The double-precision implementation is similar, but only two points are processed at a time in the inner loop. In order to hide latencies in accessing data from memory and reduce the data dependency stalls, we manually unroll the innermost loop 8 times, resulting in performing 32 angular separation calculations per loop iteration.

```
01  for (i=0; i < n1; i++) {
02      int jk = d1[i].jk;                                  // d1 is in the original data format
03      vector float x_i = spu_splats(d1[i].x);
04      vector float y_i = spu_splats(d1[i].y);
05      vector float z_i = spu_splats(d1[i].z);
06       for (j=0; j < n2; j+=4) {
07           vector float* m = (vector float*)(&d2[j]);     // d2 data has already been reordered on CPU
08           x_j =m[0];
09           y_j =m[1];
10           z_j =m[2];
11           vector float dot = spu_mul(x_i, x_j);
12           dot = spu_madd(y_i, y_j, dot);
13           dot = spu_madd(z_i, z_j, dot);
14             ……..
15        }
16  }
```

Figure 21. Dot product Cell/B.E. SPE SIMD implementation.

Once the dot product is computed, it is then converted to a bin index at which bin count needs to be updated. In the reference implementation, the bin index is obtained by comparing the computed dot product value to different bin boundaries, implemented either as a binary or sequential search algorithms. This possesses significant challenges for the SPE implementation as it requires branch instructions and results in branch divergence which is detrimental to the performance of the SIMD implementation. We solve this problem as follows (Figure 22): i) Since most of the angular separations are in the last few bins, we compare the computed dot products to the last 8 bin boundaries and mask out all but the least significant bits, resulting in a sequence of 8 zeros and ones for each dot product. ii) We then add all the masked results for a given dot product and obtain a number that specifies how many comparison tests, starting from the last bin boundary, failed, thus giving us the bin # to which the

given dot product belongs. iii) If any of the dot products turns out to be outside the last 8 bins, which is the case for about 1% of all the dot products, a similar test is performed for the next 8 bin boundaries. If this test is not sufficient, a scalar version of the comparison subroutine is invoked that performs the comparison in the same manner as in the original reference implementation.

```
// dot_v contains 4 dot products, binb_v[] is the array of bin boundaries.
// the indices_v contains the vector of reverse indices
01  cmp[0] = spu_cmpgt(dot_v, binb_v[nbins]);
02  cmp[1] = spu_cmpgt(dot_v, binb_v[nbins - 1]);
03  cmp[2] = spu_cmpgt(dot_v, binb_v[nbins - 2]);
04  cmp[3] = spu_cmpgt(dot_v, binb_v[nbins - 3]);
05  cmp[4] = spu_cmpgt(dot_v, binb_v[nbins – 4]);
06  cmp[5] = spu_cmpgt(dot_v, binb_v[nbins – 5]);
07  cmp[6] = spu_cmpgt(dot_v, binb_v[nbins - 6]);
08  cmp[7] = spu_cmpgt(dot_v, binb_v[nbins - 7]);
09  cmp[0] = spu_and(cmp[0], cmpmask);              // cpumask = {1,1,1,1}
10  cmp[1] = spu_and(cmp[1], cmpmask);
11  cmp[2] = spu_and(cmp[2], cmpmask);
12  cmp[3] = spu_and(cmp[3], cmpmask);
13  cmp[4] = spu_and(cmp[4], cmpmask);
14  cmp[5] = spu_and(cmp[5], cmpmask);
15  cmp[6] = spu_and(cmp[6], cmpmask);
16  cmp[7] = spu_and(cmp[7], cmpmask);
17  t0 = spu_add(cmp[0], cmp[1]);
18  t1 = spu_add(cmp[2], cmp[3]);
19  t2 = spu_add(cmp[4], cmp[5]);
20  t3 = spu_add(cmp[6], cmp[7]);
21  t4 = spu_add(t0, t1);
22  t5 = spu_add(t2, t3);
23  indices_v = spu_add(t4, t5)
```

Figure 22. Bin index vector implementation.

6.2.3 Histogram kernel

Once the bin to which a given dot product value belongs is found, it needs to be incremented by 1. The biggest challenge in implementing the bin updating routine is due to the fact that values in the previously computed indices vector frequently point to the same bin, introducing data dependencies. Also, SPE intrinsics do not support gathering a vector of values based on a vector of pointers. Thus, an efficient implementation of the bin update in a straightforward manner is not feasible. We address this challenge via a lookup table as demonstrated in Figure 23. We convert the computed indices vector into an integer which serves as an index of a lookup table. The lookup table value stored at this index corresponds to an array of integers which need be added to the original bin array to update the bin counts. For example, for the indices vector {1, 0, 1, 5}, the corresponding output array should be {1, 2, 0, 0, 0, 1, 0, 0} as the bin 0 (the bin indices used here are reversed compared to original bin index) will be increased by 1 (one slot in the indices vector have index of 0), bin 1 will be increased by 2 (two slots falls in bin 1), and bin 5 will be incremented by 1 (one slot has value 5). The remaining bins stay the same. The code sample that shows how this is implemented is shown in Figure 24.

The number of different indices vectors is N^4 since each slot in the indices vector can be any value from 0 to N-1. Thus, a lookup table can be created where each entry corresponds to one possible index vector. Each table entry is an array of N values. Since there is only 256KB of SPE's local store, N must be

reasonably small. We chose N to be 8, which leads to a lookup table with 4,096 entries. Each entry contains a vector of 8 short integers and each short integer is the addition value for the corresponding bin. The total size of the table is 64KB. It is pre-computed and statically stored in the local store. By rotation and logical operations, we assemble the index to the lookup table from the indices vector and bin assignment is done in one SIMD instruction once the lookup table entry is loaded.

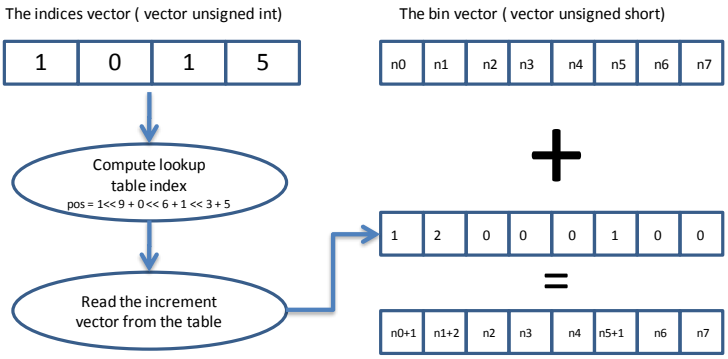

Figure 23. Histogram algorithm using a lookup table.

In the innermost loop the bin addition short integer vector is accumulated to a temporary short integer vector. It is then added to the global bins with reverse indices at the end of the outer loop iteration.

```
// this codelet accumulate addition arrays from four vectors of indices (16 normal angular separation computation)
// v0,v1,v2,v3 are vectors of indices
// large_table is the lookup table for addition values
// subtotal is the temporary vector to accumulate addition values
      vector unsigned int t0 = spu_rl(v0, 9);
      vector unsigned int t1 = spu_rl(v1, 6);
      vector unsigned int t2 = spu_rl(v2, 3);
      vector unsigned int t3 = spu_or(t0, t1);
      vector unsigned int t4 = spu_or(t2, v3);
      vector unsigned int t5 = spu_or(t3, t4);
      int indx0 = spu_extract(t5, 0);
      int indx1 = spu_extract(t5, 1);
      int indx2 = spu_extract(t5, 2);
      int indx3 = spu_extract(t5, 3);
      vector unsigned short entry0 = large_table[indx0];
      vector unsigned short entry1 = large_table[indx1];
      vector unsigned short entry2 = large_table[indx2];
      vector unsigned short entry3 = large_table[indx3];
      vector unsigned short tsum0 = spu_add(entry0, entry1);
      vector unsigned short tsum1 = spu_add(entry2, entry3);
      subtotal = spu_add(subtotal, tsum0);
      subtotal = spu_add(subtotal, tsum1);
```

Figure 24. Histogram algorithm code sample.

6.2.4 Jackknife Re-sampling

In the reference implementation, *njk* bin values are updated according to the jackknife samples computed. However, since the jackknife sample is the same for all the points processed in the innermost loop, it is sufficient to update only two bins: the bin that *jk* points to and a new auxiliary bin. The bin that *jk* points to is decreased by one and the auxiliary bin is increased by one. Thus, the bin tha *jk* points to contains the negative number and the auxiliary bin contains the total count. After all computations are done, the jackknife re-sampling can be reconstructed by add the auxiliary bin values

to all the original bins. This procedure removes the need to maintain *njk* copies of the bins for *njk* jackknife samples.

6.2.5 Double-precision implementation

The double precision implementation is similar to the single-precision implementation with adjustments made to SIMD kernels to work on 2-valued vectors instead of 4-values vectors. We report performance results for the double-precision implementation obtained on IBM QS22 blade.

6.3 Results and discussion

Performance results for Cell/B.E. and PowerXCell 8i based systems are shown in Figure 25. The application is executed using 100 random data files, with 5 bins per decade of angular scale, minimum angular separation of 1 arcminute, maximum angular separation of 10,000 arcminutes and datasets ranging from 1024 to 97K points. The execution time increases proportional to the dataset size whereas the speedup vs. AMD Opteron stays around 30× for the double-precision implementation and 50-60× for the single-precision implementation.

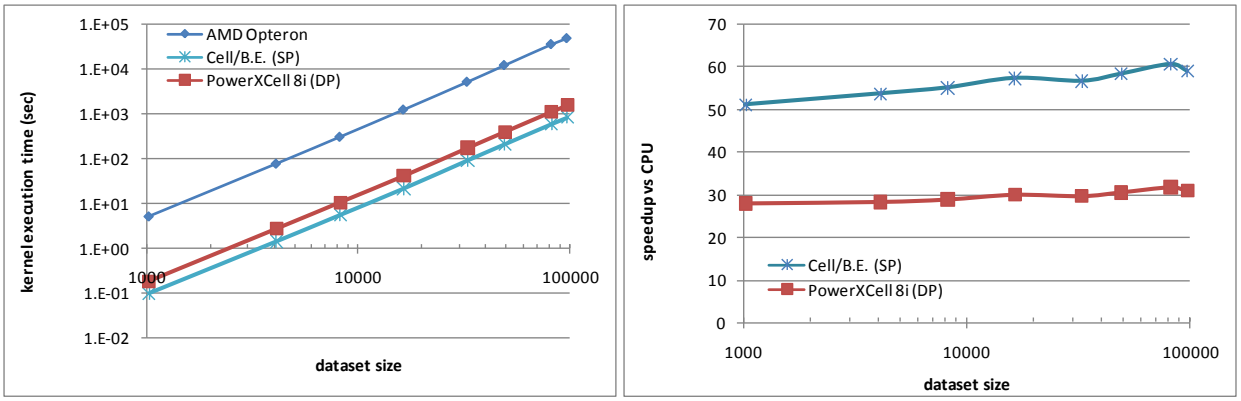

Figure 25. A comparison of the execution time (left) and speedup (right) between the AMD Opteron and Cell/B.E. (single-precision) and PowerXCell 8i (double-precision) platforms.

7 Discussion

7.1 Performance comparison across different accelerators

The goal of this project was to investigate the suitability of application accelerators, such as FPGAs, GPGPUs, etc., for speeding up the execution of computationally intensive scientific codes. In particular, we investigated implementing a cosmology application on the three most widely used accelerator architectures—FPGA, GPGPU, and Cell/B.E.—and, based on these implementations, we explored design methodologies applicable for each accelerator architecture. Figure 26 shows application performance and speedup as a function of dataset size for all systems used in this study and Figure 27 provides the final speedups achieved. It is clear that the GPU and Cell implementations of the angular correlation code provide best performance improvements whereas FPGA implementation provides only a marginal improvement. We should point out, however, that GTX-280 and PowerXCell 8i are the latest 65 nm

chips, whereas the V4 LX100 FPGA used in this study is already a couple of generations old 90 nm chip. Therefore, the comparison is not entirely fair with regards to the FPGA technology. At the same time, Nallatech H101 is a relatively new FPGA-based accelerator. This points out an issue unique to FPGA-based accelerators: there is usually a significant delay between the introduction of new FPGA chips and the availability of FPGA-based accelerators for HPC.

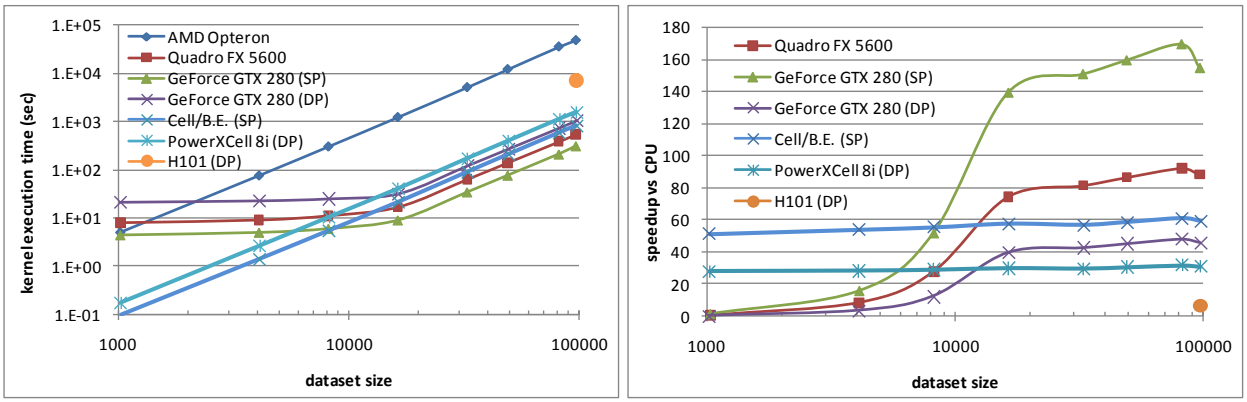

Figure 26. A comparison of the execution time (left) and speedup (right) between the AMD Opteron and 5 accelerator platforms used in this study.

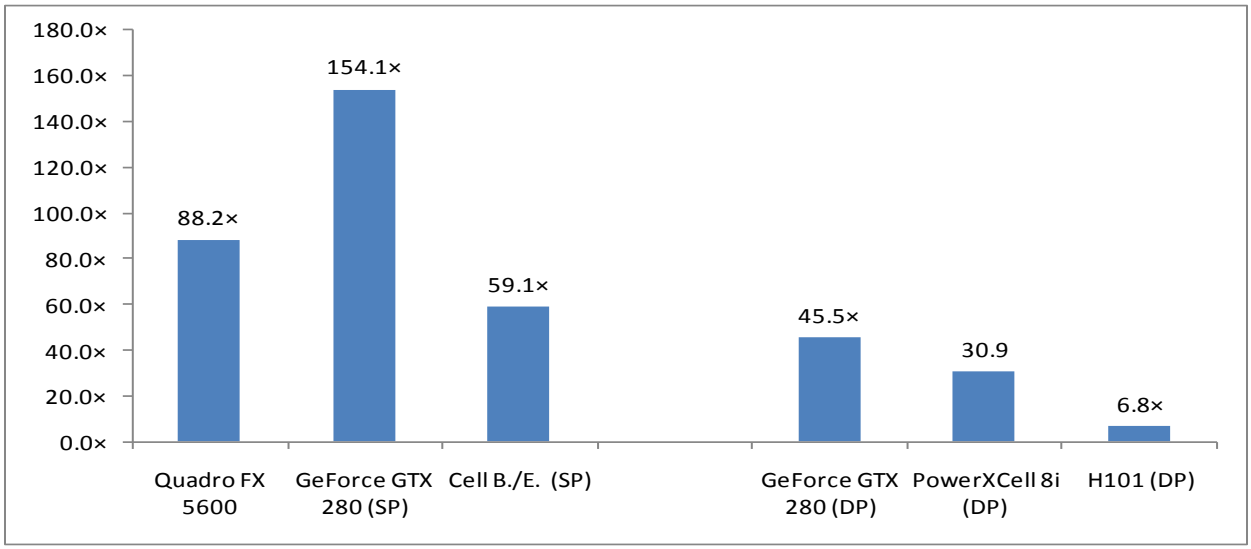

Figure 27. Speedup vs AMD Opteron.

7.2 Design methodologies

Our principal finding is that the design methodologies for these three accelerator architectures differ significantly, to the point that code portability across different accelerators is not an option. For

example, the main performance benefit for FPGA accelerators comes from pipelining the innermost loops allowing the execution of the entire body of the loop to be performed in a single clock cycle (instruction-level parallelism). Thus, the more operations that are implemented in the body of the pipelined loop, the higher the overall performance will be in terms of FLOPS. When permitted by the available space on the FPGA, we also implemented multiple instances of the compute kernel on the FPGA ("wide" parallelism). On the other hand, the GPU performance benefits come from dividing identical computations among a very large number of GPU cores (data parallelism), and taking advantage of a very high bandwidth between the cores and off-chip memory. Finally, the Cell/B.E. architecture delivers the best performance when the application can be structured as a set of independent tasks (task parallelism), each of which can be executed independently by the Cell's SPEs; yet the computations done by each SPE need to be 2- or 4-way vectorizable. This requirement places significant restrictions on the type of codes that can be efficiently implemented on the Cell/B.E. architecture. Consequently, programming models for different accelerators are very different and their efficient use by the computational science community requires specialized training, an intimate knowledge of the underlying architecture and programming tools, and concentrated development efforts.

In all cases, the principal issue when porting the kernel to an accelerator was how to break up artificial data dependencies and how to express the algorithm in a concurrent manner suitable for an execution by a large number of compute cores. Breaking up data dependencies usually requires replicating data for use by the concurrently executed kernels and merging the partial results at the end. Extracting concurrency from an algorithm is a more challenging task because of the restrictions on the instructions that can actually be executed simultaneously by a group of processors. Thus, in case of the Cell processor, the same instruction is executed on a group of 2 or 4 data elements residing in a vector and in case of the GPU, many threads execute the same instruction on different data values in a lockstep. This dictates regular data access patterns and requires eliminating branch instructions that could lead to execution sequence divergence among multiple execution treads. Since the kernel considered in this work relied on a series of branch instructions, which is a valid approach for a sequential processors that attempts to eliminate unnecessary computations, removing them was a major challenge for both the Cell and GPU implementations and was accomplished by using fairly complex lookup table based techniques. Note that in retrospective, both the FPGA and the original C implementations could benefit from such techniques.

We also investigated how the performance of computational kernels can be predicted before the kernel is ported to a given accelerator. Such performance prediction is very important in understanding the potential of accelerators prior to the expenditure of any porting effort, which, in the end might not be worthwhile. We found that generally it is relatively straightforward to predict the performance of the code on an FPGA-based accelerator, although this requires a good understanding of the FPGA-based accelerator technology; however, the performance prediction for other accelerator architectures is less straightforward. In some cases, we did find that algorithm characteristics can provide insight into the efficiency of mapping a given application kernel onto a given accelerator architecture, such as data usage patterns, data parallelism, etc.

One of the objectives of this project was to quantify the efforts necessary to implement codes on the newly emerging accelerator architectures. Based on our work we conclude that a substantial effort is required to implement the code on any of the accelerators and a specialized training is a must. Once the programmers were up to speed with the design methodology and programming tools, GPU and Cell implementations required a comparable amount of time: few days to get basic implementations that run and produce correct results, and then several months to go over multiple design alternatives and

code optimizations to achieve a good performance. FPGA implementation was time-consuming because of the long compilation time (several hours, in some cases up to a day) involved with synthesizing the application for FPGA. Thus, even to get a basic design and to verify it for correctness on an FPGA-based accelerator required a few weeks. In general, our observation is that in order to efficiently implement an application on one or another accelerator, the programmer has to fully understand the underlying algorithms used in the application and has to be prepared to rewrite these algorithms from scratch taking into account system architectures. Simple code translation is not an option if the goal is to achieve a high performance.

7.3 The big picture

Based on our experience in implementing the angular correlation and several other applications on various accelerator platforms, we attempt to provide a classification of the suitability of different types of accelerators for different HPC application classes. Figure 28 shows an attempt to assign each of the 13 dwarfs (20) to one of the accelerators considered in this study based on the underlying algorithms' suitability for a given accelerator architecture. We make the following observations:

- FPGA-based accelerators are best suited for applications with integer or fixed-point data types. Examples of such applications include signal and image processing, combinatorial logic, and many of the bioinformatics problems in which non-standard, e.g., 6 bit, integer numbers are sufficient for data representation. Floating-point codes can also be implemented and a substantial speedup can be achieved, but not as high as on GPUs and the Cell processor. Putting aside the performance issue, the main disadvantage of using FPGAs is a very time-consuming and unconventional application design cycle, whereas the main advantage is the power efficiency of FPGA-based computing.

- Cell-based accelerators are well-suited for a wide variety of compute-intensive integer and floating-point applications, particularly those that are or could be expressed as short vector computations. Examples of such applications include image processing, dense linear algebra, structured grids, and many others. The main difficulty in using Cell-based accelerators is due to the need to vectorize the code and shape the data to fit the SIMD execution model.

- GPU-based accelerators are best suited for single-precision floating-point applications with a limited amount of data dependencies. Examples of such applications include dense linear algebra, structured grid methods, spectral methods, Monte Carlo techniques, and many other types of embarrassingly parallel algorithms. Some of the double-precision applications work well on the

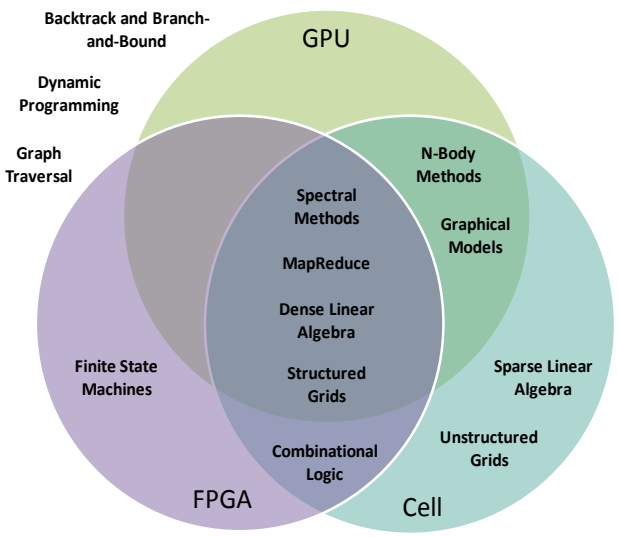

Figure 28. Mapping applications to accelerators. Applications are assigned based on the percentage of the peak performance achievable on a given architecture. The higher the achievable fraction of the peak performance, the more suitable is a given architecture for a given class of applications.

GPUs too, but the double-precision support is far behind the single-precision support on modern GPUs. The main difficulty in using GPU-based accelerators is due to the need to express the computation as a very large set of identical and independent tasks, which is not an easy process.

Applications that are control-rich, such as graph traversal, dynamic programming, and finite state machines are generally difficult to accelerate on the three types of accelerators considered in this work. They can be implemented to run on the accelerators, but they will likely achieve very little, if any, speedup. On the other hand, applications such as dense algebra and spectral methods map well to all three accelerators, meaning that a high percentage of the peak performance can be achieved on each of the architectures. However, their performance in absolute terms, e.g., FLOPS, varies significantly across different accelerator architectures. Matrix multiplication is perhaps one of the best-studies examples that demonstrates this point: (21) reports 206 GFLOPS (~60% of single-precision peak) achieved on an NVIDIA 8800GTX GPU, (22) reports achieving 202 GFLOPS (~98% of single-precision peak) on Cell/B.E. processor, and (23) report projected 29.8 GFLOPS (in double-precision) on a Virtex-5 SX240T FPGA utilizing the entire chip.

Acknowledgements

This work was supported by the NSF STCI program OCI 08-10563.

8 Bibliography

1. *Bias and variance of angular correlation functions.* **Landy, S. D. and Szalay, A. S.** 1, 1993, The Astrophysical Journal, Vol. 412, pp. 64-71.

2. *First Measurement of the Clustering Evolution of Photometrically Classified Quasars.* **Myers, A. D., et al.** 2006, The Astrophysical Journal, pp. 622-634.

3. **Nallatech.** H100 Series FPGA Application Accelerators Product Brochure. [Online] 2007. http://www.nallatech.com/mediaLibrary/images/english/5129.pdf.

4. —. FUSE: The reconfigurable computing operation system. [Online] 2002. http://www.nallatech.com/mediaLibrary/images/english/2343.pdf.

5. —. DIMEtalk V3.0 Application Development Environment. [Online] 2006. http://www.nallatech.com/mediaLibrary/images/english/4613.pdf.

6. **Xilinx.** ISE Design Suite 10.1 - ISE Foundation. [Online] 2008. http://www.xilinx.com/publications/prod_mktg/pn0010867.pdf.

7. *RAT: A Methodology for Predicting Performance in Application Design Migration to FPGAs.* **Holland, B., et al.** [ed.] V. V. Kindratenko. s.l. : ACM, 2007. High-Performance Reconfigurable Computing Technologies and Applications Workshop. pp. 1-10. 978-1-59593-894-7.

8. **Xilinx.** CORE Generator 9.1i User guide. [Online] 2008. http://www.xilinx.com/itp/xilinx8/help/iseguide/mergedProjects/coregen/coregen.htm.

9. **Nallatech.** DIME-C User Guide. [Online] 2008.

10. **Xilinx.** Xililnx DS335 Floating-Point Operator v4.0, Product Specification. [Online] 2008. http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

11. **NVIDIA.** NVIDIA GEFORCE GTX 200 GPU Datasheet. [Online] 2008. http://www.nvidia.com/docs/IO/55506/GPU_Datasheet.pdf.

12. —. CUDA 2.0 Programming Guide. [Online] 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.

13. **Podlozhnyuk, V.** 64-bin Histogram. [Online] 2007. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/histogram64/doc/histogram64.pdf.

14. *QP: A Heterogeneous Multi-Accelerator Cluster.* **Showerman, M., et al.** 2009. 10th LCI International Conference on High-Performance Clustered Computing.

15. **IBM.** IBM BladeCenter QS21 Data sheet. [Online] 2008. ftp://ftp.software.ibm.com/common/ssi/pm/sp/n/bld03006usen/BLD03006USEN.PDF.

16. —. IBM BladeCenter QS22 Data sheet. [Online] 2008. ftp://ftp.software.ibm.com/common/ssi/pm/sp/n/bld03019usen/BLD03019USEN.PDF.

17. —. Cell Broadband Engine Architecture, Version 1.02. [Online] 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/$file/CBEA_v1.02_11Oct2007_pub.pdf.

18. —. PowerXCell 8i processor product brief. [Online] 2008. http://www-03.ibm.com/technology/cell/pdf/PowerXCell_PB_7May2008_pub.pdf.

19. —. Software Development Kit for Multicore Acceleration Version 3.1 Programmer's Guide. [Online] 2008. http://publib.boulder.ibm.com/infocenter/systems/topic/eicct/prg/CBE_Programmers_Guide_v3.1.pdf.

20. **Asanovic, Krste, et al.** The Landscape of Parallel Computing Research: A View from Berkeley. [Online] December 18, 2006. http://www.gigascale.org/pubs/1008.html. UCB/EECS-2006-183.

21. **Volkov, Vasily and Demmel, James.** LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. [Online] March 13, 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.pdf. UCB/EECS-2008-49.

22. **Hackenberg, Daniel.** Fast Matrix Multiplication on Cell (SMP) Systems. [Online] March 12, 2009. http://www.tu-dresden.de/zih/cell/matmul.

23. *FPGA Based High Performance Double-Precision Matrix Multiplication.* **Kumar, Vinay B.Y., et al.** Washington, DC : IEEE Computer Society, 2009. 22nd International Conference on VLSI Design. pp. 341-346. ISSN 978-0-7695-3506-7.