Scientific Programming 17 (2009) 247–259
DOI 10.3233/SPR-2009-0286
IOS Press

Implementation of the two-point angular correlation function on a high-performance reconfigurable computer

Volodymyr V. Kindratenko [a,*], Adam D. Myers [a,b] and Robert J. Brunner [a,b]
[a] *National Center for Supercomputing Applications, University of Illinois, Urbana, IL, USA*
[b] *Department of Astronomy, University of Illinois, Urbana, IL, USA*

Abstract. We present a parallel implementation of an algorithm for calculating the two-point angular correlation function as applied in the field of computational cosmology. The algorithm has been specifically developed for a reconfigurable computer. Our implementation utilizes a microprocessor and two reconfigurable processors on a dual-MAP SRC-6 system. The two reconfigurable processors are used as two application-specific co-processors. Two independent computational kernels are simultaneously executed on the reconfigurable processors while data pre-fetching from disk and initial data pre-processing are executed on the microprocessor. The overall end-to-end algorithm execution speedup achieved by this implementation is over 90× as compared to a sequential implementation of the algorithm executed on a single 2.8 GHz Intel Xeon microprocessor.

Keywords: Reconfigurable computing, angular correlation function

1. Introduction

Correlation analyses are a common tool from the field of spatial statistics, and thereby impact a wide range of scientific disciplines. Measuring the relative clustering of occurrences within a given parameter space is of wide-ranging interest in fields from geology and paleontology [6] to genetics and epidemiology [1]. In addition, correlation functions are used extensively within the astronomy community to characterize the clustering of extragalactic objects [14,16–18]. 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. 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*).

Reconfigurable computing [7] based on the use of Field-Programmable Gate Array (FPGA) technology has evolved to the point where it can accelerate computationally intensive floating-point scientific codes beyond what is possible on conventional, microprocessor-based systems [21]. Commercially available high-performance reconfigurable computing (HPRC) platforms from XtremeData, DRC and SRC, among others, contain the hardware and tools necessary to develop and execute software that takes advantage of the fine-grain parallelism through direct FPGA hardware execution in addition to the coarse-grain parallelism available on the traditional multiprocessor systems. As a result, in the past few years considerable effort has been made to port various computational kernels to reconfigurable hardware, and to quantify and characterize their performance. However, in general, fewer attempts have been made to implement applications that go beyond a single reconfigurable proces-

*Corresponding author: Volodymyr V. Kindratenko, National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana, IL, USA. Tel.: +1 217 265 0209; Fax: +1 217 244 1987; E-mail: kindr@ncsa.uiuc.edu.

sor used as an application-specific co-processor to accelerate the computationally-intensive portion of the code. In this case study, we present a parallel implementation of a two-point angular correlation function (TPACF) algorithm on an SRC-6 reconfigurable computer in which the workload is distributed between a microprocessor and two reconfigurable processors, each consisting of two FPGAs. The main contribution of the study is a new formulation of the TPACF algorithm suitable for a parallel implementation on an FPGA-based system and its experimental validation on SRC-6 reconfigurable computer. We also project performance to multi-FPGA systems.

2. Prior work

With recent improvements in FPGA capabilities and increases in size of the chips, computational scientists began to consider this technology as a low-power high-performance alternative for conventional multiprocessors. Numerous applications – with varying degree of success – have been implemented on FPGA-based systems in the past few years. Applications that have a small, but computationally intensive kernel are usually good candidates for acceleration on FPGAs. Image and signal processing and cryptography problems are particularly well-suited for FPGAs [4,22] as these types of applications typically use integer numerical data types and are characterized by a high degree of data reuse and an execution mode in which data is streamed through a chain of processing blocks with little or no additional data produced in-between. 2D image filtering using a separable kernel [8] is just one such example in which an 18× speedup is achieved by replacing the 1D 21-point convolution kernel with an FPGA-based implementation.

Applications that require floating-point arithmetic, such as those that make an extensive use of linear algebra, FFTs, n-body particle calculations, etc., have also been ported to FPGAs. Molecular dynamics non-bonded force-field kernel is a representative example of such an application in which a 3× speedup over the conventional processor is achieved by implementing the kernel on the FPGA [12]. Performance of floating-point applications, however, is frequently limited by the resources, such as hardware multipliers, available on the FPGA chips. It is common to see floating-point applications running on FPGAs with (2–10)× speedup (when compared to modern processors) and (10–100)× speedup for applications that use integer or fixed-point arithmetic.

We first outlined the idea of using FPGAs to accelerate the computation of two-point angular correlation function in [11]. We implemented algorithm's cross-correlation kernel on SGI RASC RC100 reconfigurable processor using Mitrion-C programming language and achieved a 19× speedup as compared to Intel Itanium processor. More recently, we implemented a different version of the algorithm in DIME-C targeting Nallatech H101 FPGA accelerator add-on board achieving a 6.7× speedup over a 2.4 GHz AMD Opteron processor [9]. The work presented in this paper is based on a different FPGA platform – SRC-6 reconfigurable computer – and different software environment – SRC Computers Carte. In [15] we proposed to implement fixed-point dot product and bin mapper using Xilinx System Generator for Simulink and integrated this operator with the Carte-based application. We later found, however, that a simpler and more efficient solution is to add a custom fixed-point comparison operator written in Verilog, as described later in Chapter 6.3. We also investigated how to load-balance an FPGA-based application by partitioning the data among several FPGAs and dynamically scheduling their execution [10]. This approach allowed us to achieve a nearly 100% utilization of reconfigurable processors and boosted application performance by 9%. In the present paper, we provide a detailed description of the final algorithm and its FPGA-based parallel implementation and investigate its performance and scalability characteristics on a multi-FPGA system.

3. The SRC-6 reconfigurable computer

The SRC-6 MAPstation [19] we used consists of a commodity dual-CPU Intel Xeon board, one MAP Series C and one MAP Series E processor, and an 8 GB common memory module, all interconnected with a 1.4 GB/s low latency Hi-Bar™ switch. The SNAP™ Series B interface board is used to connect the CPU board to the Hi-Bar switch. All these components are standard.

The MAP Series C processor module contains two user FPGAs, one control FPGA and memory. There are six banks (A–F) of on-board memory (OBM); each bank is 64 bits wide and 4 MB deep for a total of 24 MB. There is an additional 4 MB of dual-ported memory dedicated solely to data transfer between the

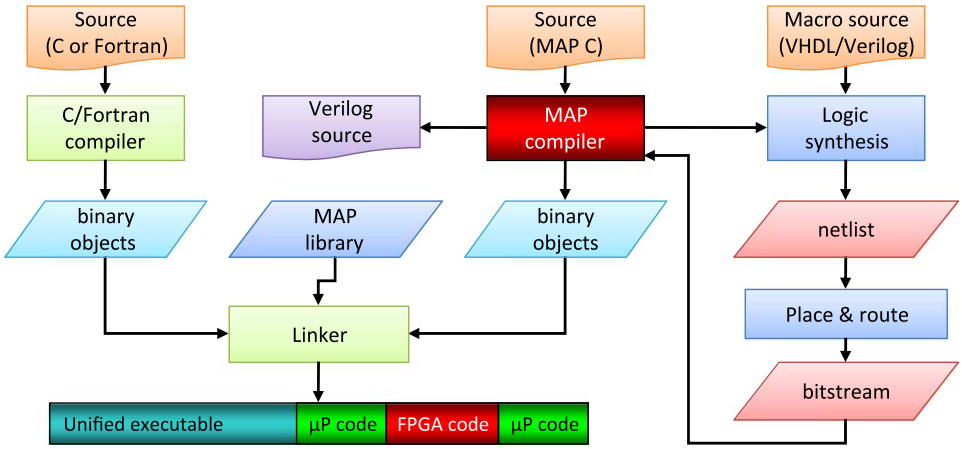

Fig. 1. Carte development flow for SRC-6 reconfigurable computer.

two FPGAs. The two user FPGAs in the MAP Series C are Xilinx Virtex-II XC2V6000 FPGAs. The FPGA clock rate of 100 MHz is set from within the SRC programming environment. The MAP Series E processor module is identical to the Series C module with the exception of the user FPGAs: the two user FPGAs in the MAP Series E are Xilinx Virtex-II Pro XC2VP100 chips.

An FPGA is a semiconductor device consisting of programmable logic elements, interconnects, and input/output (I/O) blocks (IOBs) that allow implementing complex digital circuits. Xilinx Virtex II FPGA's basic logic blocks are a four-input lookup table (LUT) and a flip-flop. Two LUTs, two flip-flops, and some control logic form a SLICE – basic building block and a unit used to measure FPGA resources utilization. Many FPGAs also include higher-level functional blocks, such as 18×18 hardware multipliers and blocks of RAM (BRAM) distributed on the chip. As an example, Xilinx Virtex-II Pro XC2VP100 FPGA contains 88,192 4-input lookup tables, 88,192 flip flops, 444 dedicated 18×18 integer multipliers and 999 KB of internal dual-ported block RAM.

Code for SRC MAPstation is written in the MAP C programming language using the Carte™ version 2.1 programming environment [20]. The Intel C (icc) version 8.1 compiler is used to generate the CPU-side of the combined CPU/MAP executable. The SRC MAP C compiler produces the hardware description of the FPGA design for our final, combined CPU/MAP target executable. This intermediate hardware description of the FPGA design is passed to Xilinx ISE place and route tools to produce the FPGA bit file. Finally, the linker is invoked to combine the CPU code and the FPGA hardware bit file(s) into a unified executable. Figure 1 presents the overall development flow using Carte.

4. The two-point angular correlation function

Estimating angular correlation functions generally requires computing histograms of angular separations between a particular set of positions in a data space [14]. The positions in question might be the set of data points themselves, histograms of angular separations for which we will denote as $DD(\theta)$, 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. Henceforth, we will often refer to $DD(\theta)$ and $RR(\theta)$ counts as "autocorrelations" and $DR(\theta)$ counts as "cross-correlations". Once such quantities are known, $\omega(\theta)$ is estimated as in [14]:

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

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 guar-

antees 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 (e.g., see [14]). 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 [14]. 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.

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 as:

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

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 k bins, meaning that there are k equally-logarithmically-spaced bins between, for example, 0.01 and 0.1 arcminutes. The bin edges are then defined by $10^{j/k}$, where $j = -\infty, \ldots, -1, 0, 1, \ldots, +\infty$, 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 bin numbers with boundaries $10^{j/k}$:

$$bin = \text{int}[k(\log_{10}\theta - \log_{10}\theta_{\text{min}})], \tag{4}$$

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

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 $\sim$0.1 arcseconds (e.g., [23]). 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*(θ) and *RR*(θ) will usually dominate this numerical imprecision.

5. TPACF algorithm

A block-diagram of the algorithm for computing TPACF is shown in Fig. 2. Initially, the data points are loaded/converted from spherical to Cartesian coordinates and the autocorrelation function, *DD*(θ), for the entire dataset is computed. Random points are then loaded/converted one set at a time. For each random set, the autocorrelation for the random dataset, *RR*(θ), and the cross-correlation between the data points and the random set, *DR*(θ), are computed. Equation (2) is applied at the end.

The computational core of the algorithm is the subroutine that calculates binned separation distributions for either *DD*(θ) (and *RR*(θ)) or *DR*(θ) style counts. The analytical binning schema presented by Eq. (4) requires the calculation of *arccos* and *log* functions, which are computationally expensive. Therefore, in practice we use a different bin mapping schema 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 $10^{j/k}$ in the *modified* ("dot product") space, θ_j, can be pre-computed as fol-

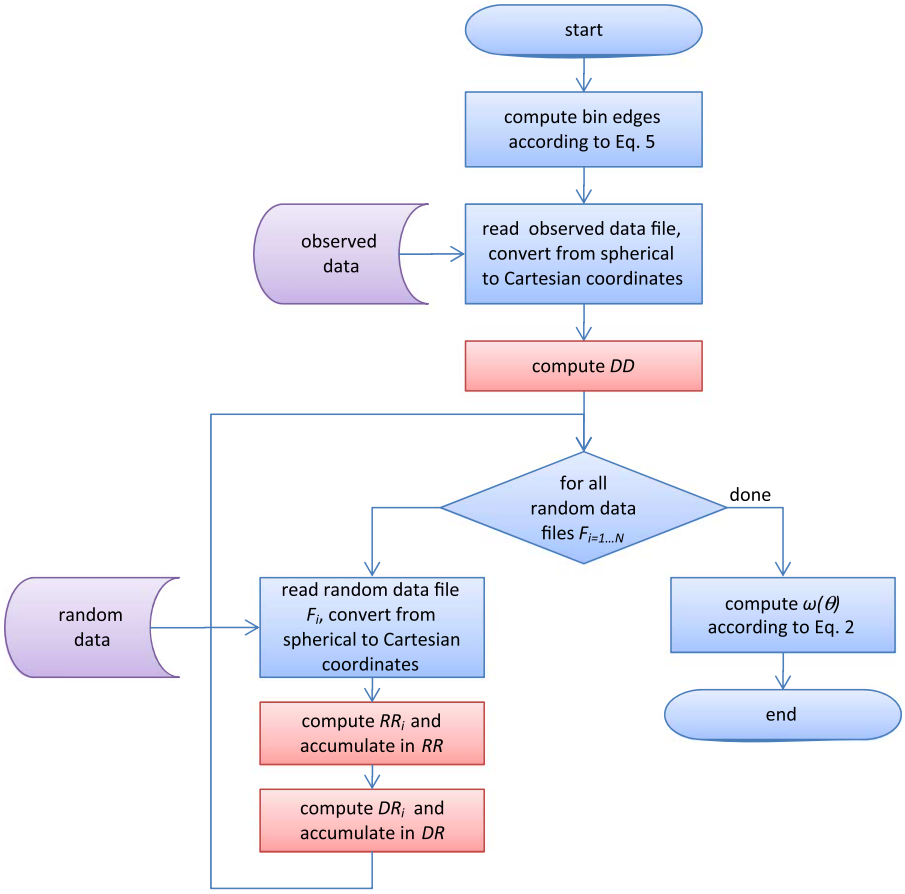

Fig. 2. Block-diagram of the TPACF algorithm.

lows:

$$\theta_j = \cos(10^{\log_{10}\theta_{\min}+j/k}), \quad j = 0, \ldots, M. \quad (5)$$

Since the bin edges are ordered, an efficient binary search algorithm [13] can be used to quickly locate the corresponding bin in just $\log_2 M$ steps (see Algorithms 1 and 2).

Computational complexity of the autocorrelation algorithm for computing binned separations is $O(N^2/2 \cdot \log_2 M)$ whereas computational complexity of the cross-correlation algorithm is $O(N^2 \cdot \log_2 M)$.

6. Implementation of the TPACF algorithm on an SRC-6 reconfigurable computer

6.1. Reference C implementation of the TPACF algorithm

The reference C implementation of the TPACF algorithm described in Section 5 is straightforward. We observe, however, that when executed on the SRC-6 host processor, only 83 MFLOPS (about 1.5% of peak floating point performance of the processor) is typically achieved. Since the binary search is applied after each dot product calculation in the main compute subroutine, performance of the reference C implementation is less dependent on the floating-point performance.

The reference C implementation was compiled using the Intel C version 8.1 compiler with compiler flags -O3, -xW and -tpp7, thus, enabling processor-specific optimizations. All code execution time measurements presented in this paper were obtained using the gettimeofday library subroutine.

6.2. FPGA implementation of the autocorrelation/cross-correlation subroutines

The reference C implementation was written with the autocorrelation and cross-correlation functions coded as a single subroutine in which autocorrelation

Input: set of Cartesian coordinates of N points $x_1, \ldots, x_N$ on the celestial sphere, set of M *modified* bins: $[\theta_0, \theta_1), [\theta_1, \theta_2), \ldots, [\theta_{M-1}, \theta_M)$;
Output: for each bin, the number of unique pairs of points (x_i, x_j) for which the dot product is in the respective bin: $B_l = |\{ij\colon\ \theta_{l-1} \leqslant \theta(x_i, x_j) < \theta_l\}|$;

1. **for** $i = 1, \ldots, N-1$ **do**
2. **for** $j = i+1, \ldots, N$ **do**
3. $d \leftarrow x_i \cdot x_j$
4. $l \leftarrow$ **bsearch**$(d, \boldsymbol{\theta})$
5. $B_l \leftarrow B_l + 1$

Algorithm 1. Autocorrelation, *DD* and *RR* counts.

Input: 2 sets of Cartesian coordinates of N points $x_1, \ldots, x_N$ and $y_1, \ldots, y_N$ on the celestial sphere, set of M *modified* bins: $[\theta_0, \theta_1), [\theta_1, \theta_2), \ldots, [\theta_{M-1}, \theta_M)$;
Output: for each bin, the number of unique pairs of points (x_i, y_j) for which the dot product is in the respective bin: $B_l = |\{ij\colon\ \theta_{l-1} \leqslant \theta(x_i, y_j) < \theta_l\}|$;

1. **for** $i = 1, \ldots, N$ **do**
2. **for** $j = 1, \ldots, N$ **do**
3. $d \leftarrow x_i \cdot y_j$
4. $l \leftarrow$ **bsearch**(d, θ)
5. $B_l \leftarrow B_l + 1$

Algorithm 2. Cross-correlation, *DR* counts.

or cross-correlation mode of execution is triggered by the input parameters. On the other hand, it can be advantageous to implement these two functions separately when porting the implementation to the SRC-6 platform because we have two different FPGA chips in the two MAPs, and each implementation can be targeted to best match the available on-chip resources to the function properties. Code executed on the microprocessor is still responsible for loading and converting data files, running the overall compute loop, computing the final results, etc. However, two versions of the computational kernel can now be outsourced to two MAPs.

6.2.1. Autocorrelation kernel

The autocorrelation subroutine was written in MAP C targeting the MAP Series C reconfigurable processor module. The design occupies both FPGAs of the MAP Series C processor and makes use of all available OBM banks. The code implemented on the primary chip is responsible for transferring bin boundaries, bin values, and the sample to be processed into OBM banks. Bin boundaries and existing bin values are transferred first; they are mirrored by each FPGA to the on-chip memory. Sample points are transferred next, they are distributed across all 6 OBM banks and permissions are given to the secondary chip to access only one half of the memory banks. The workload is then split equally between the two FPGAs. Once the entire sample of coordinate points is processed and the results are obtained on both chips, they are merged on the primary chip and streamed out to the host microprocessor.

The computational core of the autocorrelation subroutine is implemented as a nested loop that closely follows the reference C implementation, with one important exception. We note that the MAP C compiler attempts to pipeline only the innermost loop of the code. The bin search loop, used to find the bin edges that a coordinate-point-separation lies between, is the innermost loop in the reference C implementation. Pipelining this loop alone does not lead to an efficient FPGA implementation, as multiple clock cycles would have to be spent to locate the bin that needs to be updated. Therefore, it is more advantageous to fully unroll this loop and let the MAP C compiler to pipeline the *next* innermost loop instead. Thus, instead of running a binary search loop, we implement a cascade of *if/if-else* statements necessary to manually unroll the binary search loop for a fixed number of bins. This is accomplished using *selector* macro supported by SRC MAP C compiler that tests at once all bin boundaries

for a given angular separation. This way, a new result can be computed *on each iteration of the pipelined loop*, thus achieving a substantial improvement in efficiency of the overall computation. Moreover, there is enough space on each FPGA chip to unroll the loop by a factor of two. Thus, the overall execution time of this design is proportional to $N^2/8$ where N is number of points in the sample being processed: the overall algorithm complexity is $\sim N^2/2$ (autocorrelation), and the execution is split between two chips with 2 simultaneous calculations per chip.

6.2.2. Cross-correlation kernel

The cross-correlation subroutine was written in MAP C targeting the MAP Series E reconfigurable processor module. As with the autocorrelation subroutine, the code implemented on the primary chip is responsible for transferring the bin boundaries, the bin values, and the sample to be processed into OBM banks. As before, the workload is then split equally between two FPGAs, and the results are assembled at conclusion.

We introduce an extra loop in the cross-correlation subroutine in which a fraction of one of the samples being correlated is brought into the BRAM of each FPGA. This is necessary for two reasons. First, a cross-correlation, or $DR(\theta)$ count, requires two samples, a data sample and a random sample, and generally, there is not enough OBM to store the entirety of both samples. Therefore, a provision needs to be made to incorporate and process the $2N$ coordinate positions that comprise both samples as smaller subsets. This can be implemented either on the microprocessor side, or on the MAP side. We have chosen to implement this on the MAP side to minimize the penalty of calling a MAP function multiple times. Second, even if there would be enough OBM memory to store both samples concurrently, there are not enough *independent* OBM banks to provide simultaneous access to several coordinate pairs from the samples, which is needed to achieve a fully pipelined implementation. Therefore, a portion of the sample points have to be copied to several BRAM banks to provide the required memory bandwidth needed to support simultaneous calculations.

Inside the extra loop, which we introduced to divide our samples into manageable chunks, sits the rest of the code. This code is similar to the one written for the autocorrelation subroutine with two exceptions. First, the inner and outer loops are fused into a single loop. This became possible because the outer and inner loop index ranges are independent in the case of a cross-correlation. Second, MAP Series E FPGAs are larger and they have enough space to unroll the loops by a factor of three. Thus, overall execution time of this design is proportional to $N^2/6$ since there are N^2 operations to perform and they are split between two chips, each chip implementing three simultaneous calculations.

6.3. Exploiting custom-size numerical types

The difference between two smallest bin edges, $binedge_0$ and $binedge_1$, is 6×10^{-12}. Thus, just 12 digits after the decimal point (40 bits of the mantissa) are sufficient to provide the required precision used in this particular application. We experimentally verified this observation on a large data set. Thus, instead of comparing full double-precision floating-point numbers, it is sufficient to compare only the first 12 digits after the decimal point. This can be implemented in a number of ways: we can use fixed-point numerical type as in [15], or we can just scale up both the bin boundaries (on the host system before they are loaded to OBM) and the dot product (once it is computed on FPGA) by 10^{12} and use only the lower 40 bits (and the highest bit for sign) for comparison via a custom-made comparison operator written in Verilog. SRC Carte development environment provides a way to integrate third party subroutines written in VHDL or Verilog hardware description languages and therefore using the custom comparison operator was trivial. The achieved space savings were significant: over 27% of SLICEs per single bin mapping core of 31 comparison operators. As a result, the autocorrelation subroutine was extended from two simultaneous distance calculation/bin mapping cores per chip to four such cores; the overall execution time of this design is now proportional to $N^2/16$. Also, the cross-correlation subroutine was extended from three to five computational cores per chip; the overall execution time of this design is proportional to $N^2/10$. Overall resource utilization pattern has changed as compared to our previous designs: while SLICEs utilization remains almost identical, the use of hardware multipliers increased.

6.4. Exploiting task-level parallelism

Once a random data dataset is loaded from the disk, computations of the autocorrelation and cross-correlation functions involving this dataset are entirely independent and may thus be executed simultaneously. Moreover, while calculations are executed

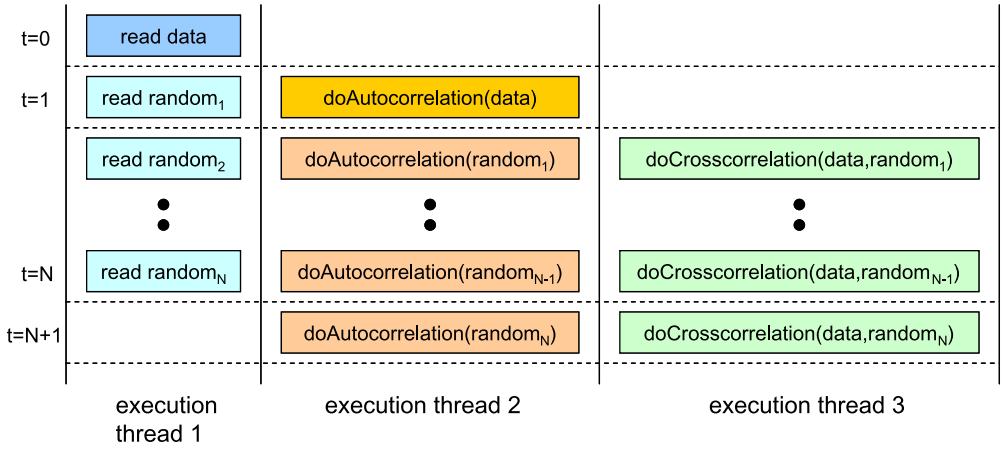

Fig. 3. The execution sequence of independent tasks in the final two-point angular correlation function implementation. Parallel scheduling of these tasks is accomplished via OpenMP.

on the MAPs using one dataset, the next random data dataset can be loaded and converted by the microprocessor. Figure 3 shows the execution sequence of different modules that can occur simultaneously. We can easily modify our reference C code to take advantage of running three simultaneous execution threads via OpenMP. One such thread is responsible for reading in a sample, the second thread is responsible for executing the autocorrelation subroutine, and the third thread is responsible for executing the cross-correlation subroutine. Only when all 3 threads are done, we move to the next random data dataset. Thus, we take advantage of the coarse-grain task-level parallelism using multithreaded execution on the conventional microprocessor platform in addition to the fine-grain instruction-level parallelism implemented via the direct hardware execution of the core algorithms on two MAPs. The overall execution time of this implementation, in accordance with Amdahl's Law, will be limited by the execution time of the slowest component, which is the cross-correlation subroutine.

For sufficiently large datasets, the measured sustained floating-point-equivalent performance of this final dual-MAP implementation is 7.4 GFLOPS, compared to 83 MFLOPs in the reference C implementation.

7. Performance and scalability

The dataset and random samples we use to calculate $\omega(\theta)$ in this work are the sample of photometrically classified quasars and the random catalogs first analyzed by [17]. We use 100 random samples ($n_R = 100$). The dataset, and each of the random realizations, contains 97,178 points ($N_D = 97{,}178$). We use a binning schema with five bins per decade ($k = 5$), $\theta_{\min} = 0.01$ arcminutes, and $\theta_{\max} = 10{,}000$ arcminutes. Thus, angular separations are spread across 6 decades of scale and require 30 bins ($M = 30$).

Figure 4 shows the percentage of the execution time expended performing various operations such as file I/O (including data conversion), computing the autocorrelation of the data, the autocorrelation of the random samples, and the cross-correlation between the data and the random samples. The overall performance of the reference implementation of the algorithm is limited by the performance of the computational core subroutine. Figure 4 (left bar) indicates that we would need to achieve a $(T_{DD} + T_{DR} + T_{RR})/T_{\text{I/O}} \approx 360\times$ speedup of this subroutine in order for our application to become file I/O (and data conversion) bound. Here T_{DD}, T_{DR} and T_{RR} are compute times for different kernels and $T_{\text{I/O}}$ is the data I/O and conversion time. While obtaining such a speedup is unlikely on the reconfigurable system used in this study, realizing even a modest speedup would lead to a substantial reduction of the overall execution time.

Figure 4 (right bar) shows the execution time percentages for our reference C implementation in which the compute kernel subroutine has been replaced with the two MAP-based subroutines, as described in Section 6.3. This result can be directly compared to that shown by the left bar, as it represents results obtained for the same sample sizes. As before, about one and a half seconds of the overall time is spent on file I/O and data conversion. However, the overall time spent by the FPGA-based autocorrelation and cross-correlation

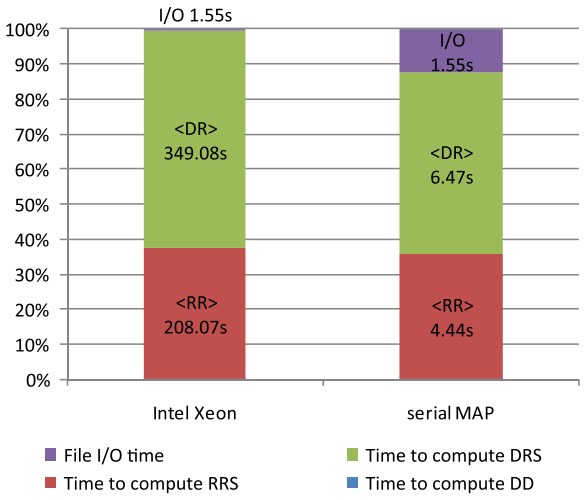

Fig. 4. Analysis of time spent while processing sample sizes of 8,000 (N_D = 8,000) points by the reference C implementation (left) and an implementation in which the compute kernel subroutine was replaced with two MAP-based subroutines (right) of the TPACF algorithm. One set of data points and 100 sets of random points (n_R = 100) of the same size were used to generate both plots. Numbers on the bars indicate the absolute execution time (in seconds) for the three major tasks. Note that time to compute *DD* counts is less that 0.4%.

subroutines is significantly smaller when compared to the microprocessor-based reference implementation. There is also a different ratio of time spent executing each of these subroutines. This is due to the differences in the implementation of these subroutines in our reference C code and MAP C codes. The bottom line is that this hybrid microprocessor/dual-MAP design outperforms the reference microprocessor-only design by a factor of $T_{\text{CPU}}/T_{\text{MAP}} \approx 44$ for samples of only 8,000 points where $T_{\text{CPU}} = T_{DD} + T_{DR} + T_{RR} + T_{\text{I/O}}$ for the CPU implementation and $T_{\text{MAP}} = T_{DD} + T_{DR} + T_{RR} + T_{\text{I/O}}$ for the FPGA implementation. Thus, even without exploiting the task-level parallelism as described in Section 6.4, we achieved a 44× speedup simply by outsourcing the execution of the two computational kernels to two MAPs used sequentially.

Figure 5(a) shows execution time as a function of the sample size for 2 implementations: the original reference C implementation and the implementation in which I/O and FPGA computations are fully overlapped, as described in Section 6.4. Note that in this implementation, $T_{\text{MAP}} \approx T_{DD} + \max(T_{DR}, T_{RR}, T_{\text{I/O}}) \approx T_{DD} + T_{DR}$ since the execution of different parts of the algorithm is fully overlapped with each other and the cross-correlation is the most time-consuming part. The execution time of the microprocessor-only reference C implementation grows quadratically with sample size, as expected. The execution time of the dual-MAP implementation increases quadratically as well, but it follows a different curve. For small datasets, overhead associated with calling the MAP-based subroutines is the dominant factor in the overall code execution time. However, as the sample size increases, this overhead becomes relatively small when compared to the time spent on actual calculations. This effect is best demonstrated in Fig. 5(b), which shows the ratio of the execution time of the reference C implementation and the dual-MAP implementation, $T_{\text{CPU}}/T_{\text{MAP}}$. This ratio indicates the relative performance improvements between different implementations. Since we take into account the time spent by each implementation to perform all the operations necessary to obtain the final set of results (bin counts in this case), it is fair to consider this ratio as the measure of the overall algorithm *speedup*. Thus, Fig. 5(b) indicates that the speedup of the dual-MAP implementation approaches 90×; in other words, our parallel implementation of the TPACF algorithm effectively achieves a performance improvement of 90× as compared to the reference C implementation.

The measurements presented in Fig. 4 (right bar) for a dataset consisting of 8,000 points show that loading the data from file and converting to Cartesian coordinates requires $T_{\text{I/O}} \approx 1.55$ seconds. Overall execution time of our dual-MAP implementation of the algorithm for the same 8,000 data points is $T_{\text{MAP}} \approx 6.63$ seconds. We can increase the computational throughput of the algorithm until the I/O becomes the dominant factor by just adding a few extra MAP pairs; in fact, just 4 MAP pairs will be almost sufficient: $T_{\text{MAP}}/T_{\text{I/O}} \approx 4$.

The number of MAP pairs, of course, depends on the size of the dataset processed. The plot in Fig. 6(a) shows dependency between the dataset size and the time it takes our parallel algorithm to execute one or another task with the corresponding dataset. Thus, the process of loading (and pre-processing) a dataset is linearly proportional to the dataset size. The overall execution time of the parallel algorithm very closely follows the execution time of the cross-correlation subroutine, which is the dominant component in the algorithm. The gap between the overall execution time and the time necessary to load the required datasets increases with the dataset size. This suggests that in order to stay I/O-bound, we would need to use additional MAP pairs as the dataset size increases. Figure 6(b) shows a projection of the number of MAP pairs that are necessary for different dataset sizes. The

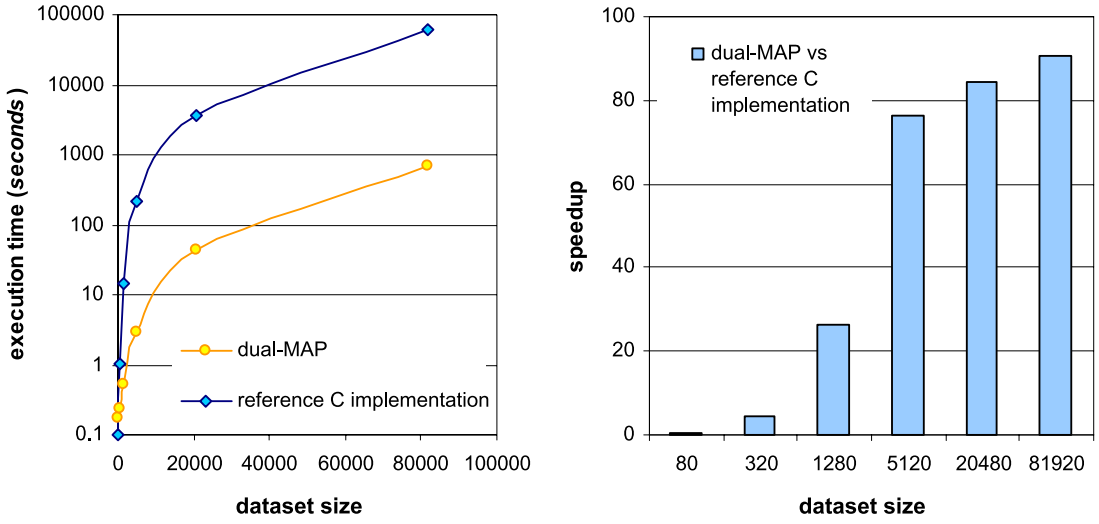

Fig. 5. (a) Execution time as a function of dataset size, (b) overall algorithm execution speedup as a function of dataset size.

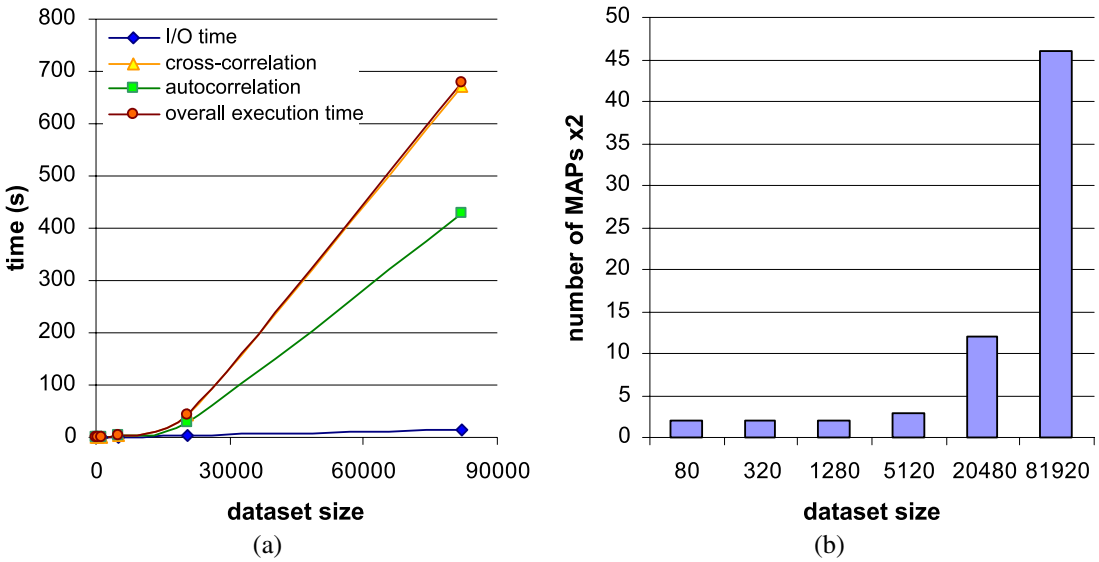

Fig. 6. (a) Execution time of various parallel threads as a function of dataset size, (b) projection of the number of MAP pairs necessary to stay I/O-limited as the dataset size increases. The projection is based on the extrapolation from the proposed model.

projected number of MAP pairs is computed as a (rounded to the next integer) ratio of the overall execution time to the time necessary to load the corresponding dataset (which we consider to be our overall execution time target). For example, overall execution time achieved on one dual-MAP system for a dataset consisting of 20,480 data points is $T_{\text{MAP}} \approx 42.49$ seconds and the time necessary to load this data from the disk is $T_{\text{I/O}} \approx 3.68$ seconds. In order for the application to become disk I/O-bound, we would need to decrease the overall execution time to be equal $T_{\text{I/O}}$ time. This can be achieved if we split the calculations among $T_{\text{MAP}}/T_{\text{I/O}} \approx 12$ MAP pairs. Adding more than 12 MAP pairs will not improve the performance as the overall execution time of the algorithm at this point will become limited by the time necessary to load (and pre-process) the data from disk.

Figure 6(b) indicates that if we attempt to stay I/O-bound, we will very quickly require a large number of MAPs. Thus, for a dataset consisting of 81,920 points

we will require 46 pairs of MAPs. With this number of MAPs, we can process the data in approximately the same amount of time as it takes to load the data from the disk. Of course in this analysis we assumed that data I/O and MAP computations are fully overlapped and CPU and Hi-Bar switch are not the bottleneck, which is the case for our dual-MAP system. These assumptions, however, may not hold on larger multi-MAP systems. In addition, we will likely encounter other system scalability issues that are not immediately apparent on a machine as small our dual-MAP system.

8. Discussion and lessons learned

Application performance improvements achieved on the dual-MAP system need to be considered in the content of system acquisition cost, power requirements, and application development efforts in order to fully understand the potential and shortcomings of reconfigurable computing. While several multi-FPGA systems are readily available, including SRC-6, their use requires a careful match between the application characteristics and the technology capabilities in order to achieve the desirable level of performance while optimally utilizing the available resources. Application development efforts necessary to transform a sequential application into a working FPGA implementation using Carte development environment are not that significant. However, development efforts necessary to arrive at an **efficient** FPGA implementation that fully utilizes all available FPGA resources are substantial. As a case in point, the following steps were involved in adapting and optimizing the TPACF algorithm for the SRC-6 reconfigurable computer:

- ***Loop pipelining*** that enables simultaneous execution of the entire loop body in a single clock cycle (fine-grain parallelism). In order to take advantage of this feature, however, we had to manually unroll the binary search loop into an equivalent of a sequence of *if/if else* statements. Once this loop is unrolled, the next innermost loop – cycling through the points in the second dataset – can be pipelined by the SRC Carte compiler. The cost of manually unrolling the binary search loop is high as it resulted in a fixed size implementation (32 bins max) and required a significant amount of FPGA random logic and routing resources, thus limiting our ability to unroll the outer loop. Yet, the advantage is significant: theoretical peak performance of the pipelined loop in our application is 500 MFLOPS (three multiplications and two additions are required to compute the dot product, running at 100 MHz and excluding data transfer overhead), or nearly 4 GOPS when counting the comparison operators.
- ***Partial loop unrolling*** that enables simultaneous execution of several loop iterations across multiple FPGAs. In order to take advantage of this feature, we had to duplicate some of the data across the multiple BRAMs on the FPGAs as well as to implement additional *control flow* logic to avoid simultaneous access to OBM banks from multiple loop body instances. The extent to which the outer loop can be unrolled is limited by the available FPGA random logic as well as the off-chip memory bandwidth necessary to sustain the fully pipelined inner loop. At the end, we were able to place only three loop body instances per chip on the MAP Series E processor, resulting in the theoretical peak performance of 3 GFLOPS, and two loop body instances per chip on the MAP Series C processor, resulting in the theoretical peak performance of 2 GFLOPS.
- ***Replacing full double-precision*** floating-point numbers with ***41-bit fixed point*** numbers and implementing a custom comparison operator resulted in a substantial reduction of the FPGA random logic and routing resources, which, in turn, enabled to implement two additional loop body instances per chip on the MAP Series E processor. This brings the theoretical peak performance of the MAP Series E processor implementation to 5 GFLOPS, and the MAP Series C processor implementation to 4 GFLOPS. While the first two techniques are rather common when optimizing applications for running on the reconfigurable hardware, this last optimization technique, while powerful, is suitable for only some applications and could be difficult to implement as it requires an in-depth analysis of the numerical characteristics of the algorithm.
- ***Scaling up*** the application to multiple reconfigurable processors (exploiting coarse-grain parallelism at the system level). This requires that the problem can be expressed as a set of independent tasks that can be executed simultaneously. In the case of the TPACF application, such independent tasks are readily available and their execution can be scheduled simultaneously, thus bringing the theoretical peak performance of the entire application to 9 GFLOPS.

Once implemented, these steps resulted in over 1,100 lines of MAP C code compared to just 50 lines of the reference C implementation of the kernel. While the code transformation and optimization techniques used by us are not specific to any particular FPGA device or system, the resulting code is not portable to FPGA-based systems other than those offered by SCR Computers because the MAP C compiler and Carte development environment only support SRC Computers line of reconfigurable processors.

While using OpenMP to orchestrate the execution of multiple tasks on the dual-MAP system was straightforward, it did not result in the even utilization of the reconfigurable processors. Remember that the overall execution time of the autocorrelation subroutine implemented on the MAP Series C processor is proportional to $N^2/16$ whereas the execution time of the cross-correlation subroutine executed on the MAP Series E processor is proportional to $N^2/10$ where N is the number of points in the dataset. Thus, MAP Series C processor finishes its work before the MAP Series E processor. We measured that in practice MAP Series C processor is idle nearly one-fifth of the time whereas MAP Series E processor is fully utilized and the actual measured performance of our final implementation is just 7.4 GFLOPS. The 1.6 GFLOPS drop from the theoretical peak performance is partially due to the non-load-balanced implementation and partially due to the data transfer and control logic overheads encountered in our FPGA implementation. Load-balanced implementation is more involved as it requires additional data partitioning and dynamic job scheduling, which is beyond the scope of this article. This issue has been addressed in a separate conference paper [10].

Since the computational complexity of our problem is O(N^2), we have not fully explored data transfer hiding techniques – another commonly used performance optimization strategy on HPRC systems. The amount of time necessary to transfer the required data from the main system memory to the OBM banks is linearly proportional to the dataset size, thus, it is responsible only for a small fraction of the overall execution time.

Reference C implementation of the algorithm is unconstrained at run-time by the number of bins and by the bin boundary values. However, this is not the case with the FPGA-based implementation. In order to achieve maximum performance, we implemented 41-bit fixed-point comparison operator which does not allow us to use bin boundaries smaller than 0.01 arcminute. In order to do so, we will need to increase the size of the numerical types used in the application. While trivial to implement, this change however will result in increased random logic utilization, which will result in reduced number of compute engines per chip, thus decreasing the overall performance. The opposite is true too: for bin boundaries that require fewer bits to represent them, we can decrease the random logic utilization, thus potentially implementing larger number of compute engines, and thus increasing the overall performance. The ability to implement arbitrary-sized numerical types and arithmetic operators is unique to FPGAs, it cannot be accomplished on any other processor technology.

Number of bins used in the application is currently hardcoded to 32. Decreasing the number of bins at runtime is not an issue as we can simply ignore unused bins. However, increasing the number of bins will require modifying the source code and will lead to increased random logic utilization.

The cost/power benefits of reconfigurable computing technology for different applications and systems have been examined extensively in [3,5]. Overall power consumed by our dual-MAP system, including the dual-CPU motherboard and hardware interfaces necessary to drive the MAPs, is about 290 Watt. A 90-CPU cluster consisting of 45 dual-Xeon nodes that theoretically could deliver the same application performance as our dual-MAP system, consumes about 9,000 Watt. Thus, the FPGA-based solution uses 3.2% of the power of the CPU-based solution. Our dual-MAP system was acquired in 2006 for about $100K, which would have been comparable to the cost of a low-end 90-CPU cluster consisting of 45 dual-Xeon nodes. It is interesting to note however that the new generation SRC-7 MAPStation with a single Series MAP H reconfigurable processor has more capability than our dual-MAP system and costs only half as much.

9. Conclusions

In this case study, we have demonstrated how a multithreaded multi-MAP application that involves computations on large datasets can be implemented on an SRC-6 reconfigurable computer. Our parallel implementation of the TPACF algorithm uses two MAPs and outperforms a similar sequential implementation executed on a 2.8 GHz Intel Xeon microprocessor by a factor of over 90. We have shown the techniques applied to the reference C implementation for transforming the code for a multi-MAP execution and demonstrated their impact on the performance. Our analysis also indicates that the computational throughput of this implementation can be increased by adding extra

MAPs until the overall execution time becomes limited by our ability to read (and pre-process) data from the disk. We have shown that the exact number of MAPs that are necessary to stay I/O-bound depends on the size of the analyzed dataset and we have provided an empirical estimate of the number of MAPs required as a function of the dataset size.

In this initial study we have investigated the performance of a naïve, brute-force O(N^2) algorithm. We realize, however, that this is not the final step. It has been demonstrated that the calculation of the separation distributions *DD*, *DR* and *RR* for N total points can be done in $N \log N$ steps using kd-tree space partitioning [16]. Our next step is to implement the kd-tree-based computational kernel and to study and compare the performance achieved by this more efficient sequential algorithm.

Acknowledgments

This work was funded by NASA grant NNG06GH15G. We would like to thank David Caliga, Dan Poznanovic, and Jeff Hammes, all from SRC Computers Inc., for their help and support with SRC-6 system. Special thanks to Trish Barker from NCSA's Office of Public Affairs for help in preparing this publication.

References

[1] J. Abello and G. Cormode, *Discrete Methods in Epidemiology: Dimacs Workshop, Data Mining and Epidemiology*, American Mathematical Society, Boston, MA, 2004.

[2] R. Brunner, V. Kindratenko and A. Myers, Developing and deploying advanced algorithms to novel supercomputing hardware, in: *Proc. NASA Science Technology Conference (NSTC'07)*, Adelphi, MD, 2007.

[3] S. Craven and P. Athanas, Examining the viability of FPGA supercomputing, *EURASIP J. Embedded Syst.* **1** (2007), 13.

[4] B. Draper, J. Beveridge, A. Bohm, C. Ross and M. Chawathe, Accelerated image processing on FPGAs, *IEEE Trans. Image Proces.* **12**(12) (2003), 1543–1551.

[5] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko and D. Buell, The promise of high-performance reconfigurable computing, *IEEE Computer* **41**(2) (2008), 78–85.

[6] L. Feyen and J. Caers, Multiple-point geostatistics: a powerful tool to improve groundwater flow and transport predictions in multi-modal formations, in: *Proc. Fifth European Conference on Geostatistics for Environmental Applications*, Neuchâtel, Switzerland, 2004.

[7] M.B. Gokhale and P.S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer-Verlag, Dordrecht, 2005.

[8] V. Kindratenko, Code partitioning for reconfigurable high-performance computing: A case study, in: *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'06)*, Las Vegas, NV, 2006.

[9] V. Kindratenko and R. Brunner, Accelerating cosmological data analysis with FPGAs, in: *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2009.

[10] V. Kindratenko, R. Brunner and A. Myers, Dynamic load-balancing on multi-FPGA systems: a case study, in: *Proc. 3rd Annual Reconfigurable Systems Summer Institute (RSSI'07)*, Urbana, IL, 2007.

[11] V. Kindratenko, R. Brunner and A. Myers, Mitrion-C application development on SGI Altix 350/RC100, in: *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, Napa, CA, 2007.

[12] V. Kindratenko and D. Pointer, A case study in porting a production scientific supercomputing application to a reconfigurable computer, in: *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Napa, CA, 2006.

[13] D. Knuth, *The Art of Computer Programming*, Vol. 3: Sorting and Searching, 3rd edn, Addison-Wesley, Reading, MA, 1997, pp. 409–426.

[14] S.D. Landy and A.S. Szalay, Bias and variance of angular correlation functions, *Astrophys. J.* **412** (1993), 64–71.

[15] D. Meixner, V. Kindratenko and D. Pointer, On using Simulink to program SRC-6 reconfigurable computer, in: *Proc. Military and Aerospace Programmable Logic Device (MAPLD'06)*, Washington, DC, 2006.

[16] A. Moore et al., Fast algorithms and efficient statistics: N-point correlation functions, in: *Mining the Sky Proc. MPA/ESO/MPE Workshop*, A.J. Banday, S. Zaroubi and M. Bartelmann, eds, Springer-Verlag, Heidelberg, 2001, pp. 71–82.

[17] A.D. Myers, R.J. Brunner, G.T. Richards, R.C. Nichol, D.P. Schneider, D.E. Vanden Berk, R. Scranton, A.G. Gray and J. Brinkmann, First measurement of the clustering evolution of photometrically classified quasars, *Astrophys. J.* **638** (2006), 622–634.

[18] P.J.E. Peebles, *The Large Scale Structure of the Universe*, Princeton University Press, Chichester, 1980.

[19] SRC Computers Inc., SRC Systems and Servers Datasheet, Colorado Springs, CO, 2005.

[20] SRC Computers Inc., SRC C Programming Environment v 2.1 Guide, Colorado Springs, CO, 2005.

[21] K. Underwood, FPGAs vs. CPUs: Trends in peak floating point performance, in: *Proc. 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 2004, pp. 171–180.

[22] T. Wollinger, J. Guajardo and C. Paar, Security on FPGAs: State-of-the-art implementations and attacks, *ACM Trans. Embed. Comput. Syst.* **3** (2004), 534–574.

[23] D.G. York et al., The sloan digital sky survey: Technical summary, *Astronom. J.* **120** (2000), 1579–1587.