Peer-reviewed version available at *Aerospace* **2016**, *3*, 28; doi:10.3390/aerospace3030028

Article

An Implementation of Real-Time Phased Array Radar Fundamental Functions on DSP-Focused, High-Performance Embedded Computing Platform

Xining Yu [1,*], Yan Zhang [1], Ankit Patel [1], Allan Zahrai [2] and Mark Weber [1]

[1] School of Electrical and Computer Engineering, University of Oklahoma, Norman, OK, USA

[2] NOAA-National Severe Storms Laboratory, Norman, OK, USA

* Correspondance: xining.yu@ou.edu

Abstract: This paper investigates the feasibility of a backend design for real-time, multiple-channel processing digital phased array system, particularly for high-performance embedded computing platforms constructed of using general purpose digital signal processors. Frist, we obtained the lab-scale backend performance benchmark from simulating beamforming, pulse compression, and Doppler filtering based on MicroTCA chassis using Serial RapidIO protocol in backplane communication. Next, a field-scale demonstrator of a multifunctional phased array radar is emulated by using the similar configuration. Interestingly, the performance of a barebone design is compared to that of emerging tools that systematically take advantage of parallelism and multicore capabilities, including Open Computing Language.

Keywords: phased array radar; embedded computing; serial RapidIO, MPAR

1. INTRODUCTION

1.1. Real-Time Large-Scale Phased Array Radar Systems

Phased array radar (PAR), especially digital PAR, ranks among the most important sensors for aerospace surveillance. A PAR system consists of three components: a phased array antenna manifold, a front-end electronics system, and a backend signal processing system. In current digital PAR systems, the radar pushes the backend system closer to the antennas, which makes the front-end system more digitalized than its analog predecessors. Accordingly, current front-end systems are mixed-signal systems responsible for transmitting and receiving radio frequencies (RF), digital in-phase and quadrature (I/Q) sampling, and channel equalization that improves the quality of signals. Meanwhile, digital PAR backend systems control the overall system, prepare to transmit waveforms, transform received data for use in a digital processor, and process data for further functions, including real-time calibration, beamforming, and target detection/tracking.

Many PAR systems demonstrate high throughput rates for data of significant computational complexity in both the front- and backends, especially in digital PARs. For example, [1] proposed a 400-channel PAR with 1 ms pulse repetition interval (PRI); assuming 8,192 range gates, each 8 bytes length in memory, in each PRI, the throughput in the front-end can reach up to 5.24 GB/s. As the requirements for such data throughput are extraordinarily demanding, at present, such front-end computing performance requires digital I/Q filtering to be mapped to a fixed set of gates, look-up tables, and Boolean operations on the field-programmable gate array (FPGA) or very-large-scale integration (VLSI) with full-custom design [2]. After front-end processing, data are sent to the backend system, in which more computationally intensive functions are performed. Compared with FPGA or full-custom VLSI chips, programmable processing devices such as digital signal processors (DSPs) offer a high degree of flexibility, which allows designers to implement algorithms in a general purpose language (e.g., C) in backend systems. For application in aerospace surveillance, target detection and tracking are thus performed in the backend. Target tracking algorithms, including the Kalman filter and its variants, predict future target speeds and positions by using Bayesian estimation

[3], whose computational requirements vary according to the format and content of input data. Accordingly, detection and tracking functions require processors to be more capable of logic and data manipulation as well as complex program flow control. Such features differ starkly from those required for baseline radar signal processors, in which the size of data involved dominates the throughput of processing [4]. As such, for tracking algorithms, a general purpose processor or graphic processor unit (GPU)-based platform is more suitable than FPGA or DSP. In sum, for PAR applications, the optimal solution is a hybrid implementation in hardware dedicated for front-end processing, programmable hardware for backend processing, and a high-performance server for high-level functions. With that knowledge, this work focuses on the front of the backend, for which we consider using DSP as the solution for general radar data cube processing.

1.2. High-Performance Embedded Computing Platforms

A high-performance embedded computing (HPEC) platform contains microprocessors, network interconnection technologies such as those of the PCI Industrial Computer Manufacturers Group and OpenVPX, and management software that allows more computing power to be packed into a reduced size, weight, and power consumption (SWaP) system [5]. Choosing an HPEC platform as a backend system for digital PAR can meet the requirements of substantial computing power and high bandwidth throughput with a smaller SWaP system or in other SWaP-constraint scenarios, including those with airborne radars. Using an HPEC platform is, therefore, the optimal backend solution.

Open standards such as Advanced Telecommunications Computing Architecture (ATCA) and Micro Telecom Computing Architecture (MTCA) [6, 7] can be used for open architectures in multifunction PAR (MPAR) systems. Such designs achieve compatibility with industrial standards and reduce both the cost and duration of development. MTCA and ATCA contain groups of specifications that aim to provide an open, multivendor architecture that seeks to fulfill the requirements of a high throughput interconnection network, increase the feasibility of system upgrading and upscaling, and improve system reliability. In particular, MTCA specifies the standard use of an Advanced Mezzanine Card (AMC) to provide processing and input–output (I/O) functions on a high-performance switch fabric with a small form factor.

An MTCA system contains one or more chassis into which multiple AMCs can be inserted. Each AMC communicates with others via the backplane of a chassis. Among chassis, Ethernet, fiber, or Serial Rapid IO (SRIO) cables can serve as data transaction media, and the number of MTCA chassis and AMCs are adjustable, meeting the requirements of scalable and diversified functionality for specific applications. Due to PAR's modularity and flexibility, its demands can be satisfied by using configurations in a physically smaller, less expensive, and more efficient way than general purpose computing center and server clusters. For this paper, we have chosen different kinds of AMCs as I/O or processing modules in order to implement the high-performance embedded computing system for PAR based on MTCA. Each of those modules works in parallel and can be enabled and controlled by the MTCA carrier hub (MCH). According to system throughput requirements, we can apply a suitable number of processing modules to modify the processing power of the backend system.

1.3. Comparison of Different Multiprocessor Clusters

Central processing units (CPUs), FPGAs, and DSPs have long been integral to radar signal processing. FPGAs and DSPs are traditionally used for front-of-backend processing such as beamforming, pulse compression, and Doppler filtering, whereas CPUs, usually accompanied by GPUs,de are for sophisticated tracking algorithms and system monitoring. As a general purpose processor, a CPU is designed to follow general purpose instructions among different types of tasks and thus allow the advantage of programming flexibility and efficiency in flow control. However, since CPUs do not accommodate for a range of scientific calculations, GPUs can be used to support heavy processing loads. The combination of a CPU and GPU offers competitive levels of flow control and mathematical processing, which enable the radar backend system to perform sophisticated algorithms in real-time. The drawback of the combination, however, is its limited bandwidth for handling data flow in and out of the system. CPUs and GPUs are designed for a server environment,

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

in which Peripheral Component Interconnect Express (PCIe) can efficiently perform point-to-point for on-board communication. However, PCIe is not suitable for high throughput data communication among a large number of boards. If the throughput of processing is dominated by the size of data involved, then the communication bottleneck downgrades the computing performance for a CPU–GPU combination. Therefore, when signal processing algorithms have demanding communication bandwidth requirements, DSP and FPGA are better options, since both can provide significant bandwidth for in-chassis communication by using SRIO while at once achieving high computing performance. FPGA is more capable than DSP of providing high throughput in real-time for a given device size and power. When the DSP cluster cannot achieve performance requirements, the FPGA cluster can be employed for critical-stage, real-time radar signal processing. However, such improved performance comes at the expense of limited flexibility in implementing complex algorithms [2]. In all, if an FPGA and DSP both meet application requirements, then DSP can be a more preferred option given its reduced cost and less complicated programmability.

2. BACKEND SYSTEM ARCHITECTURE

2.1. Overview

Typically, an HPEC platform for PAR accommodates a computing environment [4] consisting of multiple parallel processors. To facilitate system upgrades and maintenance, the multiprocessor computing and interconnection topology should be flexible and modular, meaning that each processing endpoint in the backend system needs to be identical, and its responsibility entirely assumed or shared with another endpoint without interfering other system operations. Moreover, the connection topology among each processing and I/O module should be flexible and capable of switching a large amount of data from other boards. Figure 1 shows a top-level system description of a general large-scale array radar system. In receiving arrays, once data are collected from the array manifold, each transmits and receive module (TRM) downconverts the incoming I/Q streams in parallel. To support the throughput requirement, the receivers group I/Q data from each coherent pulse interval (CPI) and send grouped data for beamforming, pulse compression, and Doppler filtering. Beamforming and pulse compression are paired into pipelines, and the pairs process the data in a round-robin fashion. At each stage, data-parallel partitioning is used to mitigate the massive amount of computations into smaller, more manageable pieces.

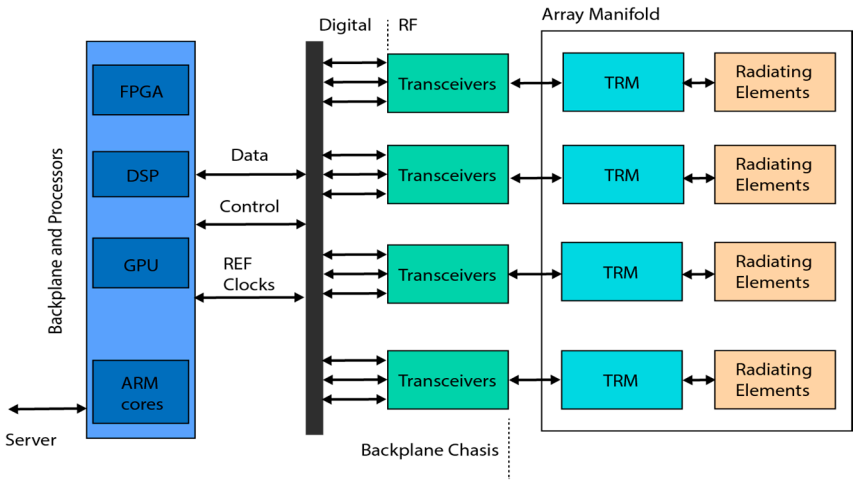

Figure 1. Top-level system digital array system concept

Fundamental processing functions for PAR (e.g., beamforming, pulse compression, Doppler processing, and real-time calibration) require teras of operation per second for large-scale PAR applications [1]. Since such processing is executed on a channel-by-channel basis, the processing flow can be parallelized naturally. A typical scheme for parallelism involves assigning computation operations to multiple parallel processing elements (PE). In that sense, from the perspective of radar application, a data cube containing data from all range gates and pulses in a CPI is distributed across

multiple PEs within at least one chassis. A good distribution strategy can ensure that systems not only achieve high computing efficiency but fulfill the requirements of modularity and flexibility as well. In particular, modularity permits growth in computing power by adding PEs and ensures that an efficient approach to development and system integration can be adopted by replicating a single PE [4]. The granularity of each PE is defined according to the size of a processing assignment that forms part of an entire task. Although finer granularity allows designers to attune the processing assignment, also poses the disadvantage of increased communication overhead within each PE [8]. To balance computation load and real-time communication in one PE, the ratio of the number of computation operations to communication bandwidth needs to be checked carefully. For example, as a PE in our basic system configuration, we use the 6678 Evaluation Module (Texas Instruments), which has eight C66xx DSP cores; an advanced configuration of that design uses a more powerful DSP module from Pro-Drive [9], which contains 24 DSP cores and four ARM cores in a single board. Texas Instruments claims that each C66xx core has 16 giga floating point operation per second (GFLOPS) at 1 GHz [10]. In our throughput measurement, the four-lane SRIO (Gen 2) link reaches up to 1,600 MB/s in NWrite mode; since the single-precision floating point format (IEEE 754) [11] occupies 4 bytes in memory, the SRIO link provides 400 million floating point data per second. The ratio of computation to bandwidth is 40 [4], meaning that the core performs up to 40 floating point operations for each piece of data that flows into the system without halting the SRIO link. As such, when the ratio reaches 40, the PE balances the computation load with real-time communication. In general, making each PE work efficiently requires optimizing algorithms, entirely using computing resources, and ensuring that I/O capacity reaches its peak.

2.2. Scalable backend System Architecture

As mentioned earlier, the features of a basic radar processing chain allow for independent and parallel processing task divisions. In pulse compression, for instance, the match filter operation in each channel along the range gate dimension can perform independently; as such, a large throughput radar processing task can be assigned to multiple processing units (PUs). Since each PU consists of identical PEs, the task would undergo further decomposition into smaller pieces for each PE, thereby allowing an adjustable level of granularity that facilitates precise radar function mapping. At the same time, a centralized control unit is used for monitoring and scheduling distributed computing resources, as well as for managing lower-level modules. PU implementations based on the MTCA open standard can balance tradeoffs among processing power, I/O functions, and system management. In our implementation, each PU contains at least one chassis, each of which includes at least one MCH that provides central control and acts as a data-switching entity for all PEs, which could be an I/O module (e.g., RF transceiver) or a processing card. The MCH of each MTCA chassis could be connected with a system manager that supports the monitoring and configuration of the system-level setting and status of each PE by way of an IP interface. Within a single MTCA chassis, PEs exchanges data through the SRIO or PCIe fabric on the backplane, and the MCH is responsible for both switching and fabric management.

Figure 2 illustrates one way to use an MTCA chassis to implement the fundamental functions of radar signal processing. Depending on the nature of data parallelism within each function, computing load is divided equally and a portion assigned to each PU. The computational capability is reconfigurable by adjusting the number of PUs, and for each processing function, a different PU can constitute at least one MTCA chassis with various types of PEs inserted into it, all according to specific needs. In the front, several PUs handle a tremendous amount of beamforming calculations, and by changing the number of PUs and PEs, the beamformer can be adjusted to accommodate different

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

types and numbers of array channels. Since computing loads are smaller for pulse compression and Doppler filtering, assigning one PU for each function is sufficient in MPAR systems.

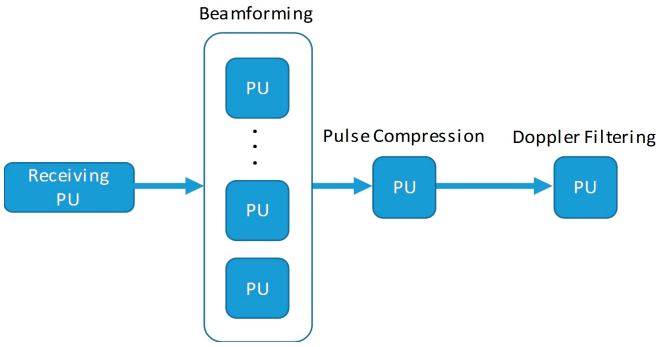

Figure 2. Illustration of the MTCA architecture in a backend

Figure 3 shows an overview of the proposed MPAR backend processing chain that focuses only on a non-adaptive core processing chain. Adaptive beamforming, alignments, and calibrations are not included in that chain until further stable results are obtained from algorithm validations. Data from the array manifold and front-end electronics are organized into three-dimensional data cubes, and N_{rg}, N_{ch}, N_p, and N_b represent the total number of range gates, channels, pulses, and beams, respectively. When any of those four numbers are red in Figure 3, the data are aligned in their corresponding dimensions. Initially, the entire data cube is divided by the number of receiving PUs that perform analog-to-digital conversion (A/D). At the output of receiving PUs, the first corner turn arranges data in the channel domain, and the number of beamforming PUs, M_b, is determined by the total number of beams divided by the number of beams that each beamforming PU can handle. Since the output of beamforming is already aligned in the range gates, such an approach can save time form the data corner turn. At the pulse compression stage, M_p represents the number of PEs, each of which in a pulse compression PU takes N_b/M_p portions of computing assignments. Prior to Doppler filtering, another corner turn reorganizes data in the pulse domain. Ultimately, the number of M_d slices of data are sent to PEs in the Doppler filtering PU.

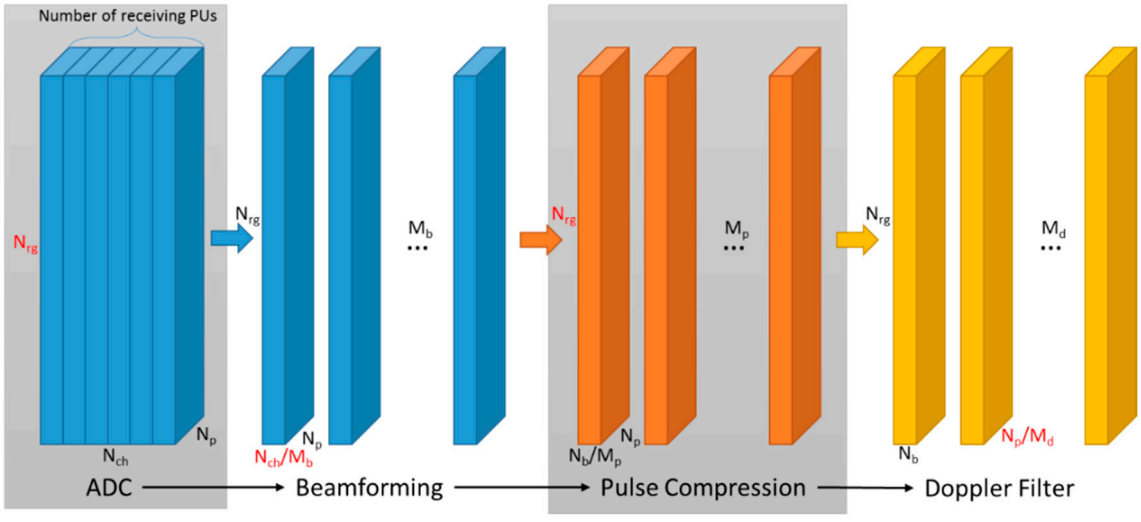

Figure 3. Overview of the non-adaptive, "core data cube" processing chain in a general digital array radar

Peer-reviewed version available at *Aerospace* **2016**, *3*, 28; doi:10.3390/aerospace3030028

2.3. Processing Unit Architecture

We currently operate an example of a receiving digital array at the University of Oklahoma (Figure 4) with two types of PUs—namely, a receiving (i.e., data acquisition) PU and a computing PU. In the receiving PU, six field-programmable RF transceiver modules [12] (e.g., VadaTech AMC518 + FMC214) sample the analog returns from TRM and send the digitalized data to a DSP module by way of an SRIO backplane. The DSP module combines and sends raw I/Q data to the computing PU through two Hyperlink ports. In the computing PU, the number of PEs is determined based on required computational loads. Moreover, each computing PU can be connected with others by way of the Hyperlink port. With that proposed PU architecture, we test the performance of the computing PU by using VadaTech VT813 as the MTCA chassis and C6678 as the PE.

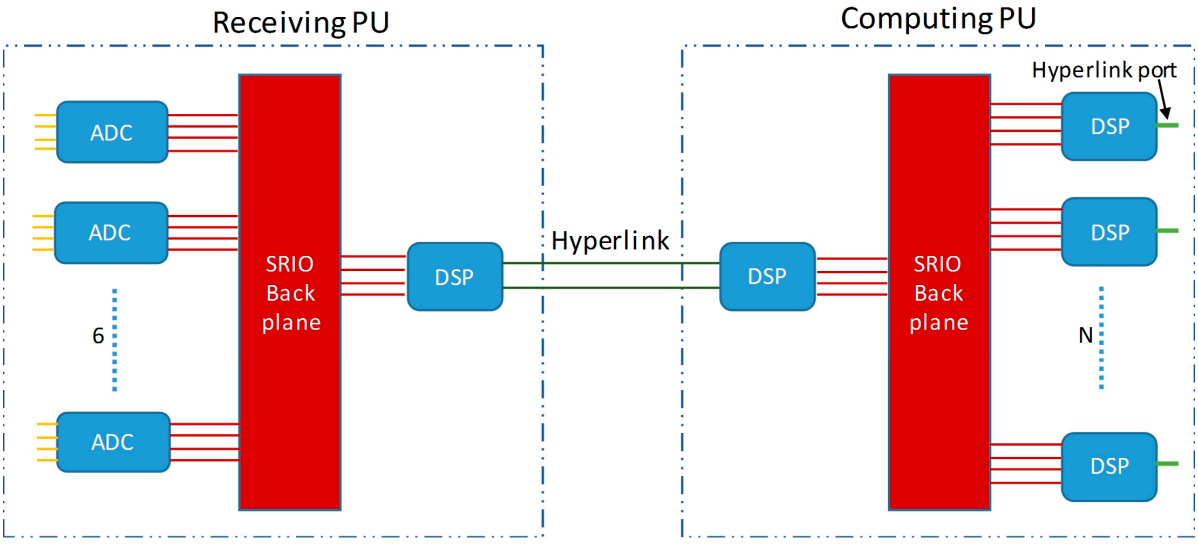

Figure 4. Simple example of a PU-based architecture

2.4. Selecting a Backplane Data Transmission Protocol

With more powerful and efficient processors, HPEC platforms can acquire significant computing power and meet scalable system requirements. However, more often than not, HPEC performance is limited by the availability of a commensurate high throughput interconnect network. At the same time, the communication overhead may be larger than the computing time, which makes the processors halt. Since that setback significantly impacts the efficiency of executing system functions, a proper implementation of the interconnection network among all processing nodes is critical to the performance the parallel processing chain.

Currently, SRIO, Ethernet, and PCIe are common options for fundamental data link protocols. RapidIO is reliable, efficient, and highly scalable; compared with PCIe, which is optimized for hierarchical bus structure, SRIO is designed to support both point-to-point and hierarchical models. It also demonstrates a better flow control mechanism than PCIe. In the physical layer, RapidIO offers a PCIe-style flow control retry mechanism based on tracking credits inserted into packet headers [13]. RapidIO also includes a virtual output queue backpressure mechanism, which allows switches and endpoints to learn whether data transfer destinations are congested [14]. Given those characteristics, SRIO allows an architecture to strike a working balance between high-performance processors and the interconnection network.

In light of those considerations, we use SRIO as our backplane transmission protocol [15], and our current testbeds are based on SRIO Gen 2 backplanes. Each PE has a four-lane port connected to an SRIO switch on the MCH. In our system, SRIO ports on the C6678 DSP support four different bandwidths: 1.25, 2.5, 3.125, and 5 Gb/s. Since SRIO bandwidth overhead is 20% in 8-bit/10-bit encoding, the theoretical effective data bandwidths are 1, 2, 2.5, and 4 Gb/s, respectively. In reality, SRIO performance can be affected by transfer type, the length of differential transmission lines, and

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

the specific type of SRIO port connectors. To assess SRIO performance in our testbed, we conducted the following throughput experiments.

Figure 5 shows the performance of the SRIO link in our MTCA test environment by using NWrite and NRead packets in 5 Gb/s, four-lane mode. Performance is calculated by dividing the payload size by the elapsed transaction time from when the transmitter starts to program SRIO registers and the receiver has received the entire dataset. First, the performance of the SRIO link is enhanced along with larger payload sizes. Second, the closer the destination memory to the core, the better the performance achieved with the SRIO link. Optimally, SRIO 4× mode can reach a speed of 1,640 MB/S, which is 82% of its theoretical link rate.

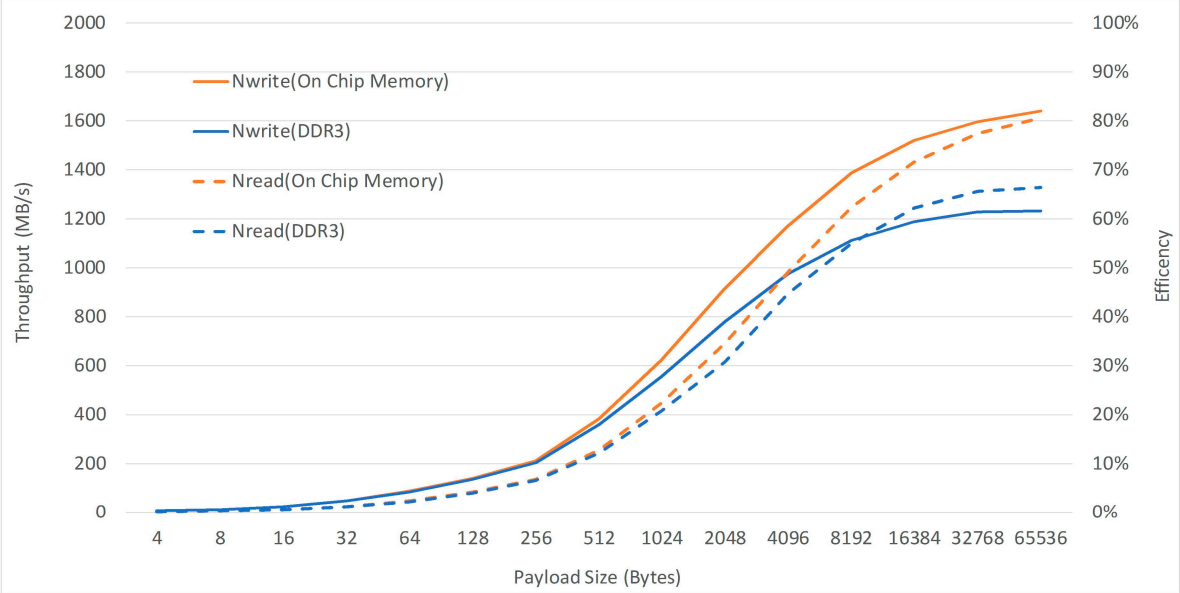

Figure 5. Data throughput experiment results in our MTCA-based SRIO testbed

2.5. System Calibration and Multichannel Synchronization

2.5.1. General Calibration Procedures

Calibrating a fully digital PAR system is a complex procedure involving four general stages (Figure 6). During the first stage, transmit–receive chips in each array channel need to calibrate themselves in terms of DC and frequency offsets, on-chip phase alignment, and local oscillator calibration. During the second stage, subarrays containing fewer channels and radiating elements are aligned precisely in the chamber environment by way of near-field measurements, plane wave spectrum analysis, and far-field active element pattern characterizations. During this stage, the focus falls upon antenna elements, not the digital backend, and initial array weights for forming focused

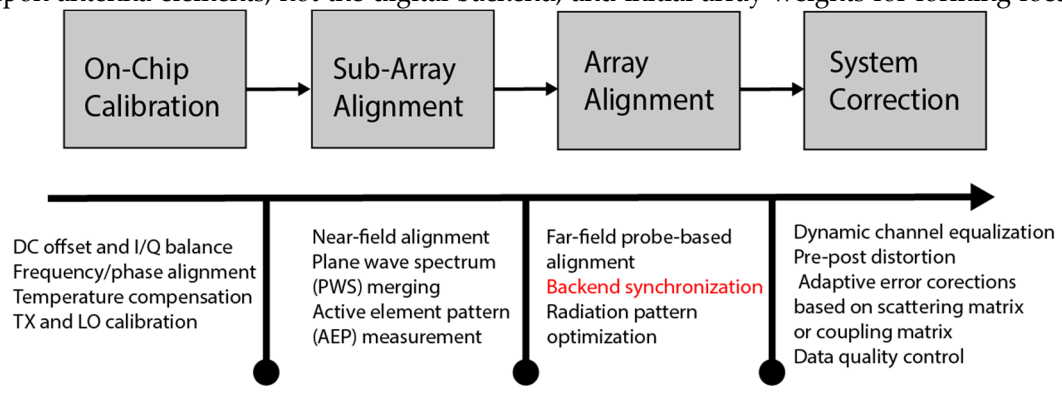

Figure 6. General system calibration procedure for DAR and the focus of this work (in red)

beams at the subarray level are estimated precisely. During the third stage, far-field full array alignment is performed in either chamber or outdoor range environments. For this stage, we use a

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

simple unit-by-unit approach to ensure that each time a subarray is added, it maximizes the coherent construction of the wavefront at each required beam-pointing direction. Array-level weights obtained at the third stage are combined with chamber-derived initial weights from the second stage to numerically optimize array radiation patterns for all beam directions. When multiple beams are formed at once, the procedure repeats for all beamspace configurations. This stage requires a far-field probe in the loop of the alignment process and requires synchronization and alignments in the backend. Initial factory alignment is finished after this stage. During the final stage, the system is shipped for field deployment, in which a series of environment-based corrections (e.g., regarding temperature, electronics drifting, and platform vibration, which is necessary for ship- or airborne radar). Based on internal sensor (i.e., calibration network) monitoring data, algorithms in the backend perform channel equalization and pre–post distortions, as well as correct system errors of deviations from the factory standard. The final step entails data quality control, which compares the obtained data product with analytical predictions to further correct biases at the data product level for desired pointing.

2.5.2. Backend Synchronization

Our study focuses only on backend synchronization during the third stage, a step necessary before parallel, multicore processing can be activated. Also, synchronized backend enables that reference clock signals in the front-end PU (and the AD9361 chips in the front-end PU) to be aligned through FPGA Mezzanine Card (FMC) interface. For the testbed architecture in Section 2.3, the front-end PU, referred to as simply "front-end" in this section, of the digital PAR systems includes a

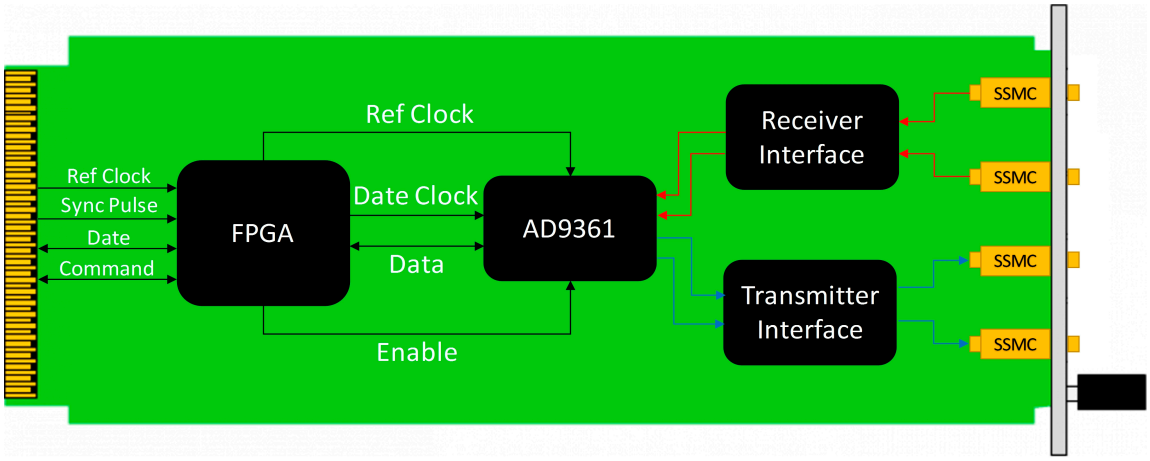

Figure 7. "PU Frontend" AMC module architecture (based on existing product used in testbed)

number of array RF channels. In each channel, there is an integrated RF digital transceiver with an independent clock source in its digital section.

Synchronization in this front-end system can be categorized according to either in-chassis or multichassis synchronization. In-chassis synchronization ensures that each front-end AMC in a chassis works synchronously with those in the other chassis. Figure 7 shows the architecture of a dual-channel front-end AMC module, which is based on an existing product from VadaTech. The Ref Clock and Sync Pulse in Figure 7 are radial fan-out by the MCH to each slot in the chassis, and each front-end AMC uses the Sync Pulse and Ref Clock to accomplish in-chassis synchronization. As an example, Figure 8 shows the timing sequence of synchronizing two front-end AMCs. Since commands from the remote PC server or other MTCA chassis may arrive at AMC 1 and AMC 2 at different times, transmitting or receiving synchronizations requires sharing the Sync Pulse between the AMCs. When AMCs acknowledge the command and detect the Sync Pulse, the FPGA triggers the AD9361 chip on both boards at the falling edge of the next Ref Clock cycle. By using that mechanism, multichannel signal acquisition and generation can be synchronized within a chassis. The accuracy of in-chassis synchronization depends on how well the trace length is matched from an MCH to each AMC. If the trace length is fully matched, then synchronization will be tight.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

For multichassis synchronization, the chief problem is so-called *clock skew*, which to overcome, requires a clock synchronization mechanism. The most common clock synchronization solution is Network Time Protocol (NTP), which synchronizes each client based on messaging with User Datagram Protocol [16]. However, NTP accuracy ranges from 5 to 100 ms, which is not precise enough for PAR application [17]. To get more accurate synchronization in the local area network, the IEEE 1588 Precision Time Protocol (PTP) standard [18] can provide sub-microsecond synchronization [19]. To implement PTP, the front-end chassis needs to be capable of packing or unpacking Ethernet packets, and additional dedicated hardware and software are required, which increase both the complexity and cost of the front-end subsystem. A better method of implementing multichassis

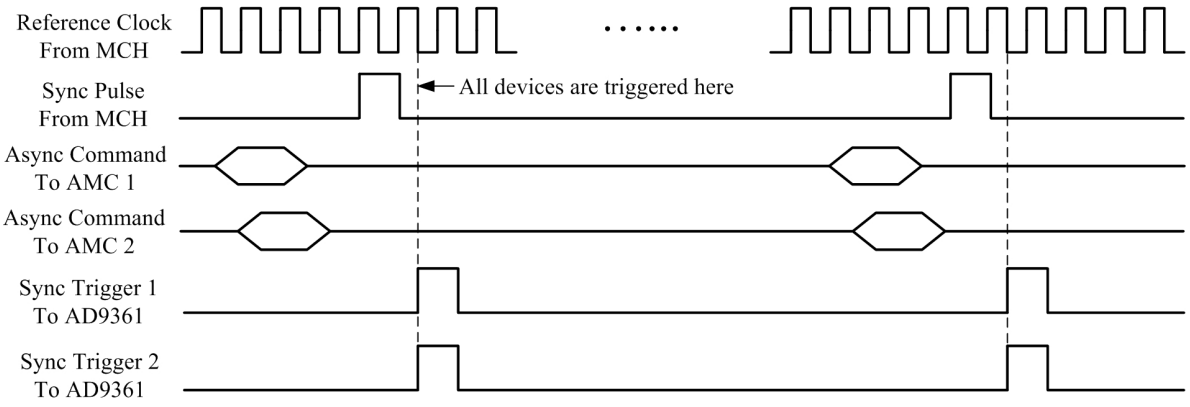

Figure 8. Frontend in-chassis synchronization timing sequence

synchronization would take advantage of GPS pulse per second (PPS), since by connecting each chassis to a GPS receiver, the MCHs can use PPS as a reference signal to generate the Ref Clock and Sync Pulse for in-chassis synchronization. Because the PPS signal among different MCHs is synchronized, the Ref Clock and Sync Pulse in each chassis is phase matched at any given time. However, when the GPS signal is inaccessible or lost, the front-end subsystem should be able to stay synchronized by sharing the Sync Pulse from a common source, which could be an external chassis clock generator or a signal from one of the chassis. In both methods, the trace length to each MCH from the common Sync Pulse source can vary, thereby making propagation time delay of the Sync Pulse from each chassis differ. To address this issue, we need to know the delay time difference of each chassis compared with the reference (i.e., master) chassis. With that knowledge, all chassis can use the time difference as an offset to adjust the triggered time.

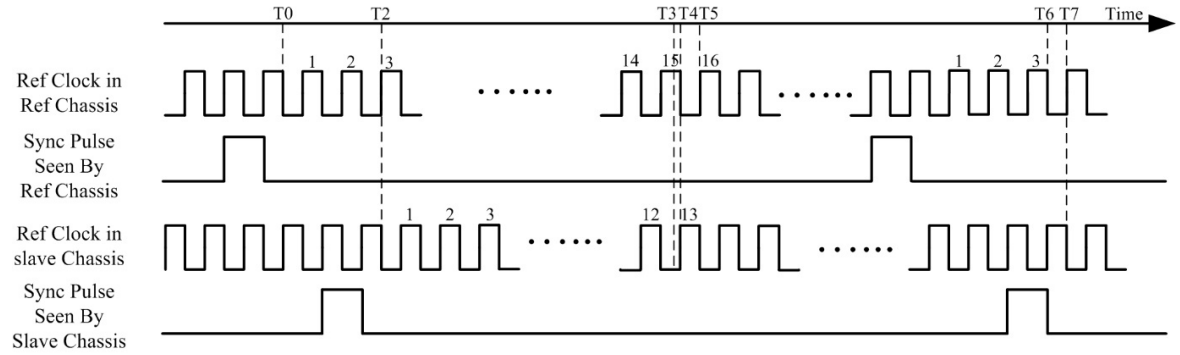

Figure 9. Example timing sequence of multi-chassis synchronization

To implement that approach, we designed a clock counter to measure elapsed clock cycles between the Sync Pulse and the return Sync Beacon, the latter of which is transmitted only from antennas connected to the reference chassis. Since the beacon arrives at all antennas simultaneously, each front-end subsystem stops its counter at the same time. The time differences in delay can be obtained by subtracting the counter number from each slave chassis to the reference chassis. Figure 9 illustrates a model timing sequence after each chassis receives the Sync Pulse. At time T0, the reference chassis begins to transmit Sync Beacon and starts the counter. After two and a half clock

Peer-reviewed version available at *Aerospace* **2016**, *3*, 28; doi:10.3390/aerospace3030028

cycles of propagation delay, the slave chassis launches the counter as well. At time T3, the Sync Beacon is received by both chassis, however, since the chassis detect the signal only at its rising edge, the reference chassis detects the signal at time T5 with counter number 16. By contrast, in the slave chassis, the counter stops at 13. In turn, when the Sync Pulse is received the next time, the reference chassis is delayed by three clock cycles and triggers AD9361 at time T6, whereas the slave chassis starts it at T7. In our example, T6 is not the same as T7. Such deviation arises because the clock phase angle between the two chassis is not identical. When this phase angle approaches 360 degrees, it is possible for the Sync Beacon to arrive when the rising edge of one clock has just passed, while the rising edge of the next clock cycle is still approaching. In the worst-case scenario, only one clock cycle synchronization error will occur, meaning that the accuracy of multichassis synchronization refers to the period of the reference clock. One way to enhance its accuracy is to reduce the period of the reference clock; however, the sampling speed of ADC confines the shortest period of the clock, because a front-end AMC cannot read new data in every clock cycle from ADC when AMC's reference clock frequency exceeds ADC's sampling speed. In our example, since the maximum data rate in AD9361 is 61.44 million samples per second, the interchassis synchronization accuracy without using the GPS signal is 16 ns.

2.6. Backend System Performance Metrics

To measure the benchmarks of digital PAR backend system performance, millions of instructions per second is often used as the metric. Meanwhile, to measure the floating point computational capability of a system, we use GFLOPS [20]. To evaluate the real-time benchmark performance, we simulate the complex floating point data cubes. For parallel computing systems, parallel speedup and parallel efficiency are two important metrics for evaluating the effectiveness of parallel algorithm implementations. Speedup is a metric of latency improvement for a parallel algorithm compared with a serial algorithm distributed over M PUs, defined as:

$$S_M = T_S/T_P \tag{1}$$

In Equation (1), T_S and T_P are the latency of the serial algorithm and the parallel algorithm, respectively. Ideally, we expect $S_M = M$, or perfect speedup, although such is rarely achieved in practice. Instead, parallel efficiency is used to measure the performance of a parallel algorithm, defined as

$$E_M = S_M/M \tag{2}$$

E_M is usually less than 100%, since the parallel components need to spend time on data communication and synchronization [4], also known as *overhead*. In some cases, overhead is possible to overlap with computation time by using multiple buffering mechanisms. However, as the number of parallel computing nodes increases, the data size of each computing node lessens, meaning that the computing nodes would need to switch between processing and communication more often, thereby inevitably resulting in what is known as *method call overhead*. When the algorithm is distributed across more nodes, such overhead can preclude the benefit of using additional computing power. Parallel scheduling thus needs to minimize both communication and method call overhead

3. Real-Time Implementation of Digital Array Radar Processing Chain

3.1. Beamforming Implementation

The procedure of beamforming is to convert the data from channel data (range gate) to beamspace, steer the radiating direction, and suppress the sidelobes by applying the beamformer weight, W_i, to received signal, Y_i, indicating in Equation (3). In the nonstationary conditions, adaptive beamforming is necessary to synthesize high gains in beam-steering direction and reject/minimize energy for other directions. Here we assume the interference environment as well as the array radar system itself is stable and does not change dramatically, hence the beamforming weights do not need

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at *Aerospace* **2016**, *3*, 28; doi:10.3390/aerospace3030028

to be update rapidly and provided offline from the external server. Any adaptive weight computing, (e.g. adaptive calibration), is not included in this study. The benefit of off-line computing is the flexibility of modifying and developing in-use adaptive beamforming algorithms.

Table 1. Equation parameters

C	Number of channels obtained by each PU
B	Number of beams processed by each PE
M	Numbers of PUs
N	Numbers of PEs in a PU
$\boldsymbol{\Omega = M \times C}$	Total number of receiving channels
$\boldsymbol{\Theta = N \times B}$	Total number of beams
$\boldsymbol{W_i^{\Theta}}$	Number of B weight vectors for the i[th] receiving channel
$\boldsymbol{Y_i}$	The i[th] receiving channel

$$Beam^{\Theta} = \sum_{i=1}^{\Omega} W_i^{\Theta} Y_i \,, where\ W_i^{\Theta} = \bigcup_{k=1}^{N} W_i^{(\mathrm{k-1})\mathrm{B}+1} \tag{3}$$

$$Beam^{\Theta} = \left[\sum_{i=1}^{C} W_i^{\Theta} Y_i\right] + \left[\sum_{i=1}^{C} W_{C+i}^{\Theta} Y_{C+i}\right] + \left[\sum_{i=1}^{C} W_{2C+i}^{\Theta} Y_{2C+i}\right] + \cdots\cdots + \left[\sum_{i=1}^{C} W_{(M-1)C+i}^{\Theta} Y_{(M-1)C+i}\right] \tag{4}$$

$$Beam^{\Theta} = \sum_{j=0}^{M-1} \left[\sum_{i=1}^{C} W_{jC+i}^{\Theta} Y_{jC+i}\right] \tag{5}$$

$$\sum_{i=1}^{C} W_{mC+i}^{\Theta} Y_{mC+i} = \bigcup_{k=1}^{N} \left(\sum_{i=1}^{C} W_i^{(\mathrm{k-1})\mathrm{B}+1} Y_i\right) \tag{6}$$

Typically, multiple beams pointing to different directions are formed independently in the beamforming process. A straightforward implementation is to providing a number of beamformers to form concurrent beams in parallel. Since each beamformer requires the signal from all the antennas, the data routing between antennas and beamformer would become complex when the number of channels is large. To reduce the routing complexity, as showing in Equation (4, 5), the entire data is divided equally and a portion assigned to each sub-beamformers, (i.e. computing node), in which the term $\sum_{i=1}^{C}\left(W_{jC+i}^{b} Y_{jC+i}\right)$ is calculated independently. A formed beam is generated by accumulating the results from each sub-beamformers. This method is named as *systolic beamforming* [21].

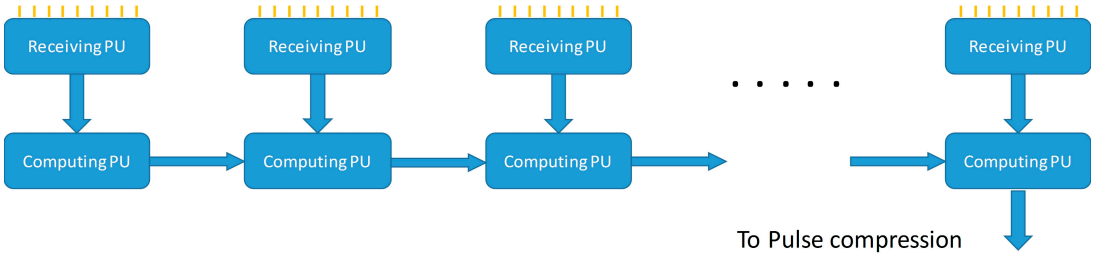

Figure 10. Scalable beamformer architecture

In our implementation, the received data from Ω channels are sent to a number of *M* PU, in which, as showing in Equation (6), each PE calculates the term $\sum_{i=1}^{C} W_i^{(\mathrm{k-1})\mathrm{B}+1} Y_i$ to form *B* partial beams in parallel. After all the PEs finish the computing, one PU starts to pass the result to its lower neighbor, in which the received data are summed with its own and the results are send downstream. In turn, after the last PU combined all the results, and the entire number of Θ beams based on Ω channels are formed. Based on the PU shown in Figure 4, a scalable beamforming system architecture is represented in Figure 10.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

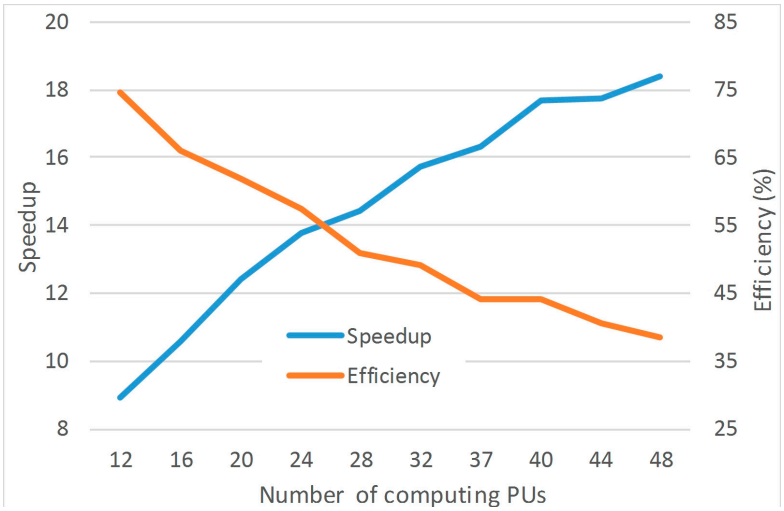

Figure 11. Speedup and efficiency of beamforming implementation

In the Equation (3), there are one multiplication and one addition for each range gate. Since each complex multiplication and addition require six and two real flops respectively, the computing complexity of the proposed real-time beamforming is $(6+2) \times N_c \times N_{rg} = 8N_cN_{rg}$, where N_c and N_{rg} are the number of channel and range gates. For given processing time interval T, the throughput of beamformer is $(8N_cN_{rg})/T\ FLOPS$. Figure 11 shows the performance of beamforming by using different numbers of PUs as an example, in which $N_c = 480$ and $N_{rg} = 1024$. In this figure, the speedup grows with the number of PUs, but the efficiency is degradation due to the reason of method call overhead. As for this reason, we need to seek a balance between the performance and effectiveness based on the system requirements. According to Figure 11, an optimal choice, for example, M = 28, allows the system to achieve a good speedup while maintain a reasonable level of efficiency.

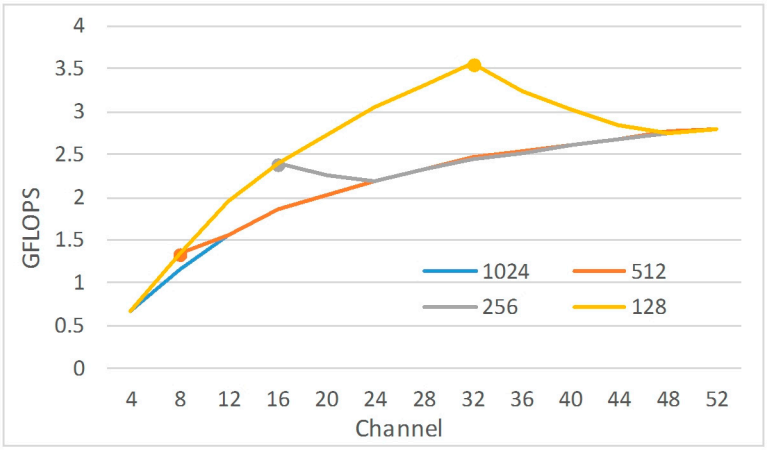

Figure 12. DSP core performance versus the number of range gates

Capacity cache miss is another issue affecting the performance of real-time beamforming. A cache miss occurs when the cache memory does not have sufficient room to store the data used by the DSP core. For example, in Figure 12, when channel number equals to 16, if there are no cache misses, four cases should have the same number of GFLOPS. However, for the cases that the numbers of range gates equal to 128 and 256, the beamformer can outperform than the cases that range gates are 512 and 1024. This variation is caused by cache miss. The markers in Figure 12 represent the maximum number of channels that the DSP cache memory can hold for a specific number of range gates. Before reaching each maker point, the performance improvement of each case is from using larger sizes vectors, which reduces the method call overhead. However, after reaching the maker points, the benefit of using large sizes of the vectors is compromised by the cache misses.

Fortunately, the capacity cache miss can be mitigated by splitting up data sets and processing one subset at each time, which is referred as *blocking* or *tiling*. In that sense, the data storage is handled carefully so the weight vectors will not be evicted before the next subset reuses it. As an example, one DSP core forms 15 beams from 24 channels, and each channel contains 1024 range gates, so W_i^b and Y_i in Equation (3) are the matrices of dimensions 24×15 and 24×1024, which are 3 KB and 192 KB. As the size of L1D cache is 32KB, to allow the weight vectors and input matrix fitting into L1D cache, the data from 24 channels should be divided into 16 subsets. So, one large size beamforming based on 1024 range gates is converted into 16 small size beamforming based on 640 range gates. The beamforming performance of this example is listed in Table 2, in which the performance of DSP core remains the same regardless the size of input data. Note that the performance shows in Table 2 is based on one C66xx core in C6678. Since we already considered the I/O bandwidth limited when all eight core works together, C6678, or similar multi-core DSP, performances can be deduced from multiplying the number in Table 2 by the number of DSP cores.

Table 2. DSP core performance after mitigating cache misses

GFLOPS: Range Gates (Subsets) / Channel	1024 (16)	512 (8)	256 (4)	128 (2)
4	0.67	0.67	0.67	0.67
8	1.34	1.34	1.34	1.33
12	1.97	1.96	1.96	1.95
16	2.42	2.42	2.41	2.40
20	2.81	2.81	2.80	2.78
24	3.15	3.15	3.14	3.12
28	3.45	3.44	3.43	3.40
32	3.71	3.70	3.68	3.66
36	3.92	3.91	3.87	3.82
40	4.10	4.09	4.04	3.96
44	4.25	4.24	4.19	4.08
48	4.39	4.38	4.34	4.23
52	4.49	4.48	4.46	4.35

3.2. Pulse Compression Implementation

The essence of pulse compression is matched filtering operation, in which the correlation of the return signal, $s[n]$, and replica of the transmitted waveform, $x[n]$ is performed. Matched filter implementation converts the signal into the frequency domain, point-wise multiplies with a waveform template, and then converts the result back to time domain [4], as shown in Equation (7-10). Since the length of FFT in Equation (7-8) needs to be the first power of 2 greater than $N + L - 1$, zero padding $x[k]$ and $s[k]$ are necessary. As zero padding increases the length of input vectors, the designer should properly select the values of N and L to avoid unnecessary computation.

$$S[k] \overset{FFT}{\longleftarrow} s[n] \quad 0 \le n \le N \tag{7}$$

$$X[k] \overset{FFT}{\longleftarrow} x[n] \quad 0 \le n \le L \tag{8}$$

$$Y(k) = S[k]X[k] \; 0 \le k \le (N + L - 1) \tag{9}$$

$$y[n] \overset{IFFT}{\longleftarrow} Y[k] \quad 0 \le n \le (N + L - 1) \tag{10}$$

Based on Equation (7-10), the computing complexity of pulse compression depends on FFT, IFFT, and point-wise vector multiplication. In radix-2 FFT, there are $log_2 N$ butterfly computation stages, in which it consists of $N/2$ butterflies. Since each butterfly requires one complex multiplication, one complex addition, and one complex subtraction, the complexity of computing radix-2 FFT is: $C_{FFT} = (6 + 2 + 2) \times (N/2) \times log_2(N) = 5Nlog_2(N)\ flops$. Because the computing complexity of IFFT is the same as FFT, the throughput of the pulse compression in frequency domain

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

is $(2 \times C_{FFT} + NC_{mult})/T = (10Nlog_2(N) + 6N/T)\ FLOPS$, in which N is the number of range gates after zero padding, and C_{mult} is the complexity of point-wise complex multiplication.

The computation throughput of pulse compression and FFT measured on one C66xx core is in Figure 13, in which dots represent the maximum number of range gates that the L1D cache can hold. It is evident that the calculation performance would degrade dramatically when the data size is close to or over the cache size, and the performance of pulse compression and FFT correlates to each other. Similar to beamforming, the pulse compression performance in a multi-core or multi-module system can be calculated by multiplying the throughput showing in Figure 13 by the number of cores enabled.

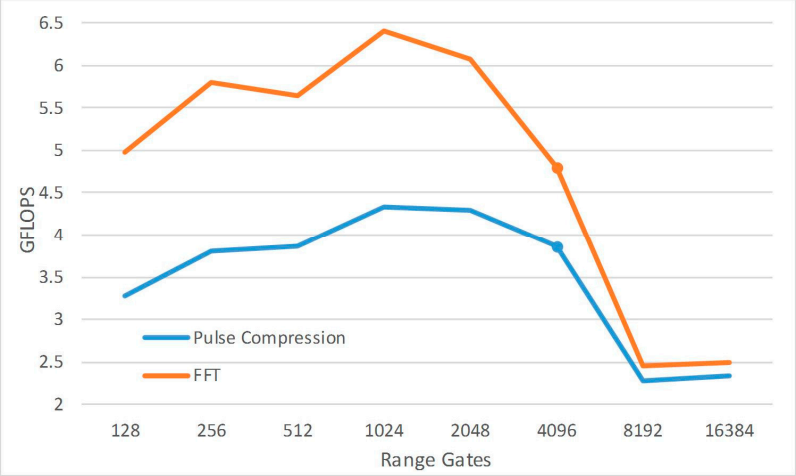

Figure 13. Performance of pulse compression and FFT vs. differences sizes of range gates

3.3. Doppler Processing and Corner Turn

The first objective of the Doppler processing is to extract moving target sfrom stationary clutters. The second objective is to measure the radial velocity of the targets by calculating the Doppler shift [22], from the Flourier transform of data cube along the CPI dimension. Hence, the throughput of the Doppler filter is $C_{FFT}/T = 5N_p log_2(N_p)/T\ FLOPS$, where N_p is the number of pulses in one CPI. In our environment, the throughput per core of Doppler filter is shown in Table 3. Again, hardware-verified performance in FLOPS linearly increases with the number of DSP cores. As the output of the pulse compression is arranged along the range gate dimension, the output needs to undergo a corner turn before being handled by the Doppler filtering processors [23]. This two-dimensional corner turn operation is equivalent to a matrix transpose in the memory space. Using EDMA3 [24] on TI generic C66xx DSP, the data can be reorganized into the desired format without interfering the real-time computations in DSP core. Table 4 shows the performance of data corner turn by using EDMA3 under different conditions.

Table 3. Doppler filtering performance per core

GFLOPS: Pulses / Range Gate	8	16	32	64	128
1024	0.7293	1.6036	2.6852	3.8543	4.2866
2048	0.7294	1.6000	2.6841	3.8543	4.2867
4096	0.7294	1.5999	2.6842	3.8544	4.2867
8192	0.7295	1.6000	2.6842	3.8544	4.2732

4. Performance Analysis of Complete Signal Processing Chain

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at *Aerospace* **2016**, *3*, 28; doi:10.3390/aerospace3030028

In the previous sections, we measured the computing throughput of each basic processing stage in our backend architecture, in which the performance for both communication and computing are sensitive to the size of data involved. Based on previous discussions and inspired by a future full-size MPAR type system, we use the overall array backend system parameters in Table 5 as an example.

Table 4. Time consumption of corner turn for one beam

Time (μs) Pulses / Range Gate	8	16	32	64	128
1024	21	33	64	126	253
2048	40	66	130	254	502
4096	135	265	513	1011	2028
8192	528	1025	2033	4070	8107

Table 5. Example Digital Array Radar System Parameters

Parameters	Value	Depends on
Range gates	4096	Pulse Compression
Pulses	128	Doppler Filtering
Channels per chassis	48	Beamforming
Beams per PE	22	Number of channels per chassis
PRI	1 *ms*	Pulse compression computing time
CPI	128 *ms*	$PRI \times number\ of\ pulses$
No. of beamforming PU	16	Total number of antennas required by application
No. of PE in each PU	12	Total number of beams required by application
Total No. of Beams	264	$PE \times number\ of\ beams\ per\ PE$
Total No. of Channels	768	$PM \times number\ of\ channels\ per\ chassis$
Total No. of PU	18	Beamforming + Match Filter + Doppler Processing

In our study, the entire backend processing chain is treated as one *pipeline*. Similar to a traditional FIFO memory, the *depth* of the pipeline represents the number of clock cycles or steps between the first data flow in the pipeline until the result data comes out. In Table 5, the critical parameter is the number of range gates. As shown in Figure 13, when the number of range gates is 4098, the pulse compression performance is well-balanced. Based on this range gate number, we can estimate the processing time of pulse compression. This latency confines the shortest PRI that the backend system allows for real-time processing. Based on the parameters in Table 5, the *time scheduling* of the radar processing chain is shown in Figure 14. This scheduling is a rigorous and realistic timeline including

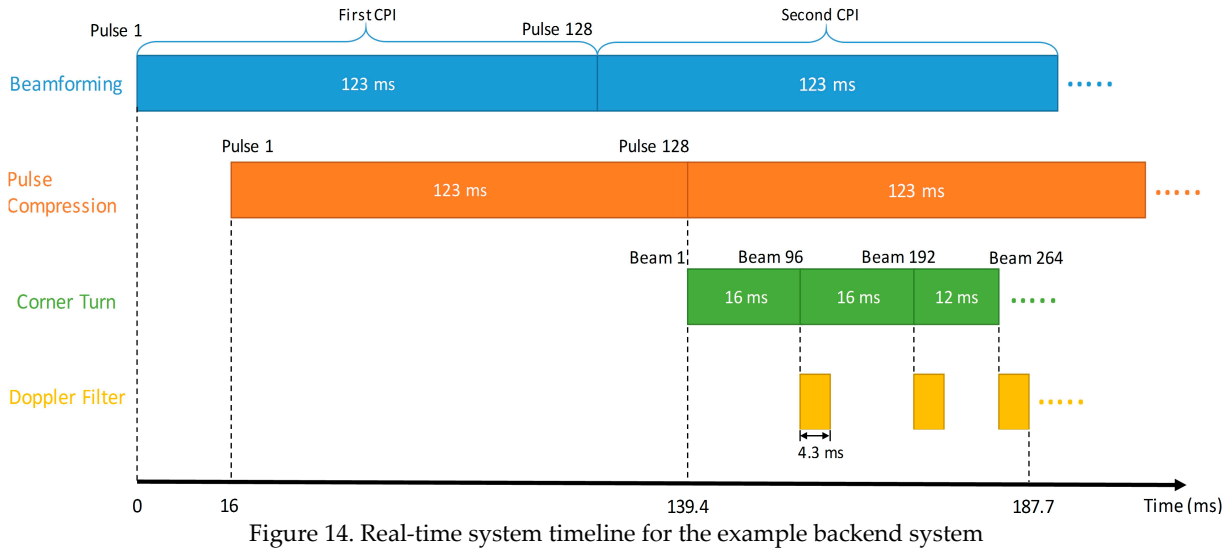

Figure 14. Real-time system timeline for the example backend system

all the impacts of SRIO communication and memory access, and has been verified by real-time hardware running tests. The numbers of PU and PE are chosen as an example, which can be changed based on different requirements. The overall latency, depth of pipeline, for the backend system is **1.5 CPI or 187.7 *ms***. Firstly, the parallel beamforming processors use 123 *ms* to generate 264 beams for the 128 pulses in one CPI. After all the beamforming results are combined to the last PU and sent to the pulse compression processors, the pulse compression takes another 123 *ms*. For the Doppler processing, in 16 *ms*, the first 96 beams from 128 pulses will be realigned in the CPI domain and sent out to the Doppler filter. In total, there are 192, 12, and 12 number of DSPs are involved for the beamforming, pulse compression, and Doppler filter, respectively. And for each processing function, it achieves 6880 GFLOPS, 370 GFLOPS, and 140 GFLOPS real-time performance, respectively.

5. Parallel implementation using Open CL/Open MP

The previous section summarizes the approach of "manual task division and parallelization." Another option is using standard and automatic parallelization solutions. For example, *OpenCL* is a standard for parallel computing on heterogeneous devices. The standard requires a host to dispatch tasks, or kernels, to devices which perform the computation. To leverage OpenCL, the 66AK2H14 is loaded with an embedded Linux kernel that contains the OpenCL drivers. The ARM cores run the operating system and dispatch kernels to the DSP cluster. For systems with more than one DSP cluster, OpenCL can dispatch different kernels to each cluster. Kernels must be declared in the host program. Because OpenCL is designed for heterogeneous processors that do not necessarily share memory, OpenCL buffers are used to pass arrays to the device. When the kernel is dispatched, arrays must be copied from host memory to device memory. This communication adds significant *overhead* to computation time that increases linearly with buffer size. The K2H platform does share memory between host and device, so the host can directly allocate memory in OpenCL memory, or CMEM.

The DSP cluster registers only as a single OpenCL device, so once it has received the kernel, the computation must be distributed across the DSP cores. There are two options for distributing the workload. OpenCL can distribute computation among the DSP cores, but this requires all cores to execute same programs. This distribution limits flexibility and complicates algorithm development. The second option is OpenMP. OpenMP is a parallel programming standard for homogenous processors with shared memory. The OpenMP runtime dynamically manages threads during execution. These threads are forked out from a single master thread when it encounters a parallel region in the program. These areas are denoted with OpenMP pragmas (`#pragma omp parallel`) in C and C++. This mechanism allows us to incorporate parallel regions into generally serial code very easily. This ease of use comes with a performance penalty. OpenMP spends processor time on managing threads so the coder must take care to minimize the number of threads in their program. This overhead can be overcome by allocating larger workloads to each thread.

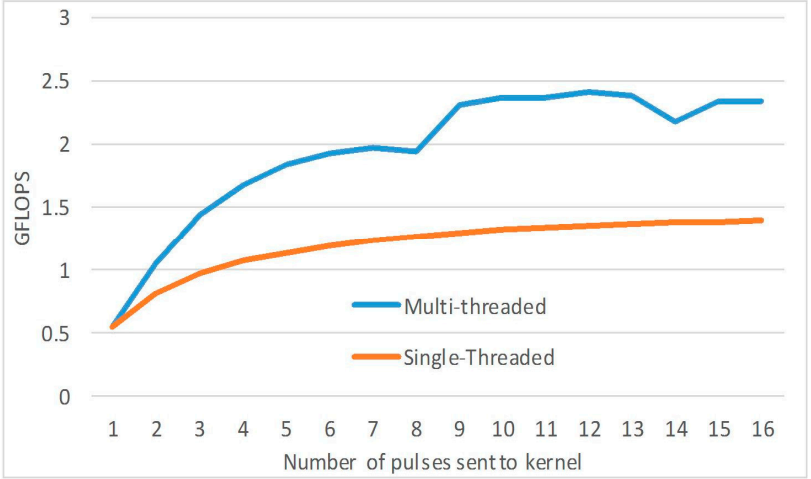

Figure 15. Beamforming kernel performance using OpenCL (8192 rang gates)

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

The DSP cluster registers only as a single OpenCL device, so once it has received the kernel, the computation must be distributed across the DSP cores. There are two options for distributing the workload. OpenCL can distribute computation among the DSP cores, but this requires all cores to

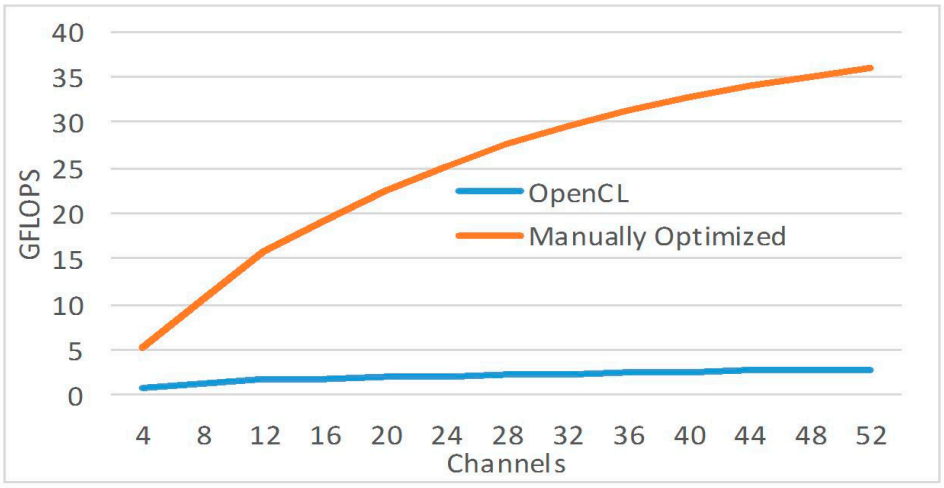

Figure 16. Comparing OpenCL performance to manually optimized code (8192 range gates) for beamforming

execute same programs (similar to the way a GPU operates). This distribution limits flexibility and complicates algorithm development. The second option is OpenMP. OpenMP is a parallel programming standard for homogenous processors with shared memory. The OpenMP runtime dynamically manages threads during execution. These threads are forked out from a single master thread when it encounters a parallel region in the program. These areas are denoted with OpenMP pragmas (#pragma omp parallel) in C and C++. Ideally, a coder would write code that can easily be computed in a serial manner (i.e. on a single thread), and then add in the OpenMP pragma to parallelize the task. This mechanism allows us to incorporate parallel regions into serial code very easily, and also allows easy removal of OpenMP regions. A prime use case for OpenMP is for loops.

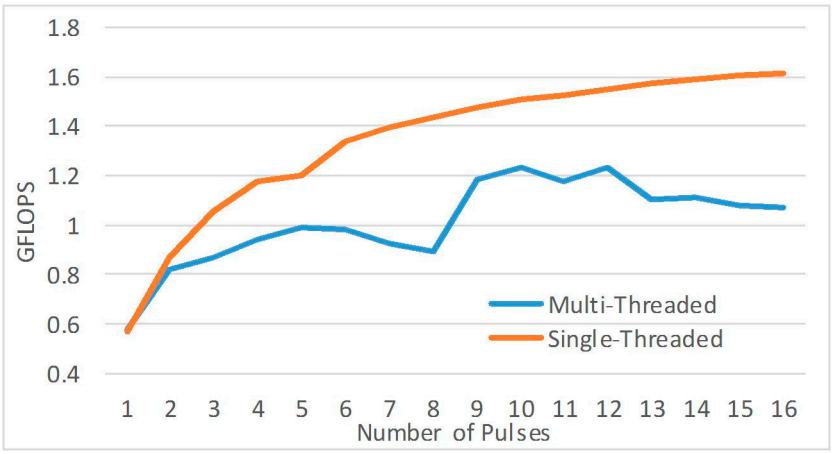

Figure 17. Pulse compression performance using OpenCL (8192 range gates)

If the result of each iteration is independent of all the others, the OpenMP pragma #pragma omp parallel for can be used to dynamically distribute the each iteration to its own thread. Without OpenMP, a single thread will execute each iteration one-by-one. With OpenMP, iterations are distributed among the threads in real-time and are executed in parallel. Each thread handles one iteration at a time until the OpenMP runtime detects the end of the loop. This ease of use comes with a performance penalty. OpenMP spends processor time on managing threads so the coder must take steps to minimize OpenMP function calls. This overhead can be overcome by allocating larger workloads to each thread. For example, if doing vector multiplication, the coder should allocate blocks of the vectors to each thread instead of assigning a single multiplication to each thread. Another penalty to take into consideration is memory accesses. With multiple threads trying to access

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

(in most cases) non-contiguous memory locations, the overhead can be increased drastically. This is part of the reason why most multi-core system's performance does not scale linearly with the number of processors. This effect can clearly be seen Figure 19 below.

5.1. Beamforming Implementation with OpenCL/OpenMP

For beamforming, the kernel processes an arbitrary number of beams. The processing of each beam is allocated to its OpenMP thread. Figure 15 shows that as the number of beams sent to the kernel increases, the time it takes to process an individual beam decreases. Because of OpenMP

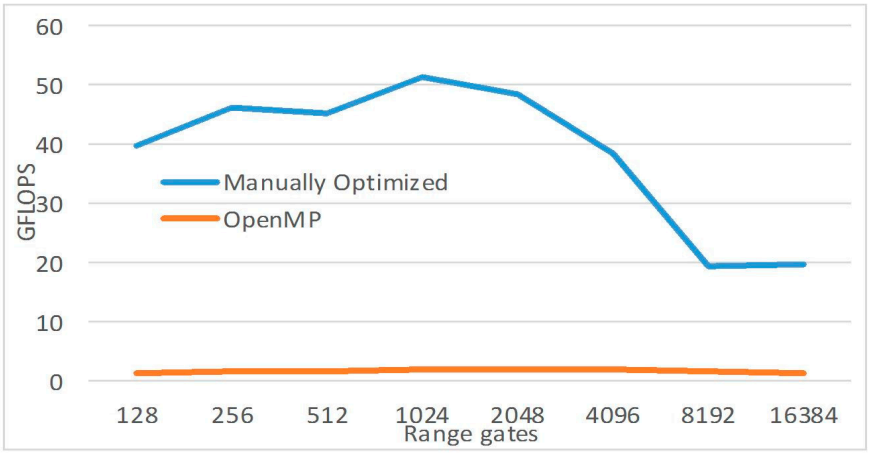

Figure 18. Comparing OpenCL performance to manually optimized code for pulse compression

overhead, the performance of the kernel increases logarithmically.

Comparing the performance of OpenCL/OpenMP implementation to the manually optimized scheme, the overhead of standard scheme can be seen more clearly in Figure 16. On average, using OpenCL/MP results in a 33% average performance penalty in beamforming with a maximum penalty of 44.3% when performing beamforming from 48 channels.

5.2. Pulse Compression Implementation with OpenCL/OpenMP

In pulse compression, once again the kernel can receive an arbitrary number of pulses. Each beam is processed in its OpenMP thread. However, in this case, multi-threaded execution is not favorable as shown in Figure 17. This difference is due to the highly non-linear memory accesses required by the FFT and IFFT. This effect is more pronounced for large sized FFTs. When multiple

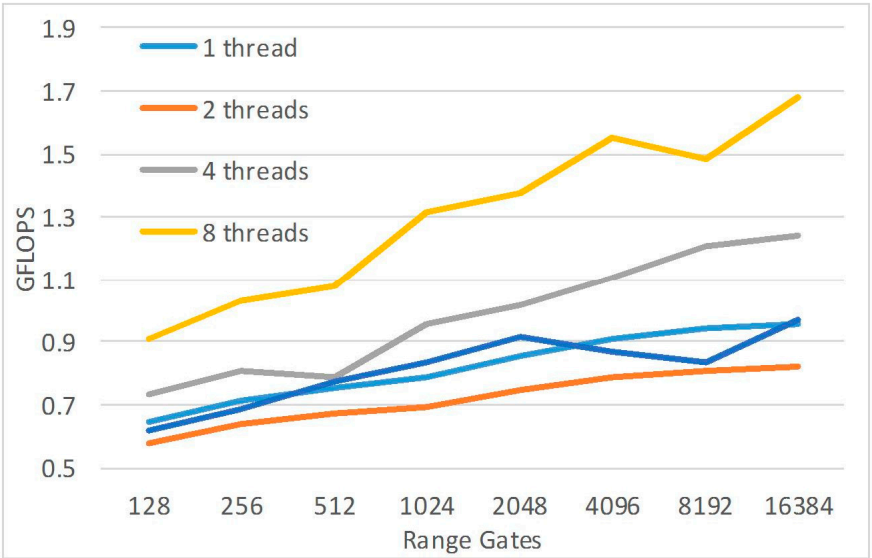

Figure 19. Doppler processing performance using OpenCL

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 23 July 2016 doi:10.20944/preprints201607.0071.v1

Peer-reviewed version available at Aerospace 2016, 3, 28; doi:10.3390/aerospace3030028

FFTs and IFFTs are running in parallel, the non-linear accesses are compounded which results in severely degraded performance.

The comparison in Figure 18 shows the performance of OpenCL/MP implementation compared with the manually optimized codes. It should be noted that the L1 cache loading optimizations were not used in the kernel as L1 cache cannot be used as a buffer in OpenCL kernels.

5.3. Doppler Processing Implementation with OpenCL/OpenMP

The Doppler processing kernel is set up differently from the previous two steps due to the large amount of small FFTs to be done. In this kernel, we configure the number of threads manually. Each

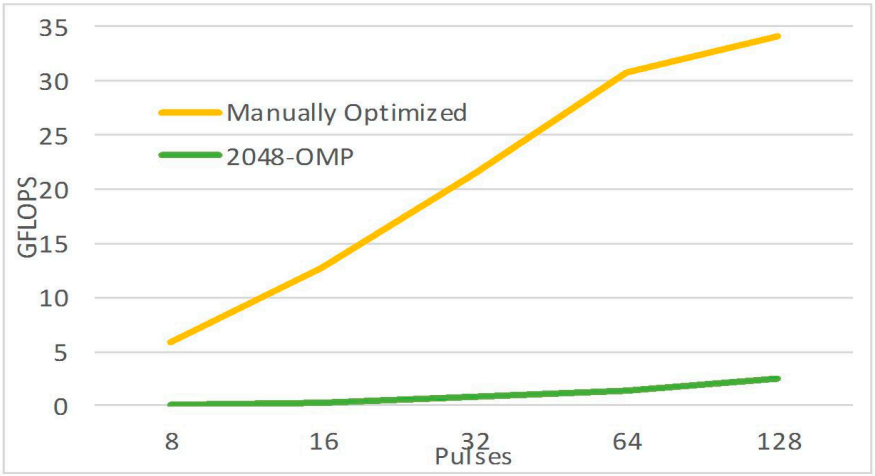

Figure 20. Comparing OpenCL performance to manually optimized code for Doppler processing

thread is allocated a fixed portion of the data to process. The number of threads must be a power of 2 so that an even amount of data is sent to each one. It is possible to set any number of threads that is a power of 2, but if that number exceeds the number of physical cores, OpenMP and loop overhead begin to degrade overall performance. Figure 19 shows the performance of the kernel for different numbers of threads with varying numbers of range gates, and Figure 20 compares the performance of manual optimization scheme with OpenCL/MP scheme.

6. SUMMARIES

In this study, we present a development model of an efficient and scalable backend system for digital PAR based on Field-Programmable-RF channels, DSP core, and SRIO backplane. The architecture of model allows synchronized and data-parallel real-time surveillance for radar signal processing. Moreover, the system is modularized for scalability and flexibility. Each PE in the system has a proper granularity to maintain a good balance between computation load and communication overheads.

Even for the *basic* radar processing operations studied in this work, teras-scale floating point operations are required in the MPAR type backend system. For such requirement, using software programmable DSP that can be attuned to the processing assignment in parallel would be a good solution. The computational aspects of a 7400 GFLOPS throughput phased array backend system have been presented to *illustrate* the analysis of the basic radar processing tasks and the method of mapping those tasks to an MTCA chassis and DSP hardware. In our implementation of PAR backend system, the form-factor can be changed based on requirements of various systems. By changing the number of PUs, the total capacity of the system can be easily scaled. By changing the number of inputs for each PE, we can adjust the throughput performance of a PU. A carefully customized design of different processing stages in DSP core also helps to achieve the optimal performance regarding latency and efficiency. When we parallelize a candidate algorithm, there are two steps in the design process. First, the algorithm is decomposed into several small components. Next, each algorithm component is assigned to different processors for parallel execution. In the parallel computing, the

communication overhead among parallel computing nodes is a key impact on the *parallel efficiency* of the system. Within each parallel processor, dividing the entire data cube into small subsets to avoid cache miss is also necessary when the size of input data is larger than the cache size of processors. For data communication links, the SRIO, HyperLink, and EDMA3 handle the data traffic between and/or within each DSP. By using SRIO, the data traffic among DSPs can be switched through the SRIO fabric controlled by an MCH of the MTCA chassis, which is more flexible than PCIe and efficient than Gigabit Ethernet. A novel advantage of our proposed method is utilizing EDMA3 and Ping-pong buffer mechanism, which helps the system to overlap the communication time with computing time and reduce the processing latency. OpenCL is a framework to control the parallelism in high level, in which the master kernel assigns the tasks to each slave kernels. Compared with the barebones method of paralleling algorithms in DSP, OpenCL is platform independent and enables heterogeneous multicore software development, which leads to the drawback of less customized and efficient to specific hardware.

Acknowledgment: This research is supported by NOAA-NSSL through grant #NA11OAR4320072. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Ocean and Atmospheric Administration.

Reference

[1] J.C. Mark Weber, James Flavin, Jeffrey Herd, Michael Vai, Muti-function Phased Array Radar for U.S. Civil-Sector Surveillance Needs, DOI (2005).
[2] D. Martinez, T. Moeller, K. Teitelbaum, Application of Reconfigurable Computing to a High Performance Front-End Radar Signal Processor, The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology, 28 (2001) 63-83.
[3] L.D. Stone, R.L. Streit, T.L. Corwin, K.L. Bell, Bayesian Multiple Target Tracking, Second ed., Artech House, Norwood, MA, 2013.
[4] D.R. Martinez, R.A. Bond, M.M. Vai, High performance embedded computing handbook : a systems perspective, Boca Raton : CRC Press, Boca Raton, 2008.
[5] A. Castillo Atoche, J. Vazquez Castillo, Dual Super-Systolic Core for Real-Time Reconstructive Algorithms of High-Resolution Radar/SAR Imaging Systems, Sensors, 12 (2012) 2539-2560.
[6] P.I.C.M. Group, AdvancedTCA® Base Specification, 2008.
[7] P.I.C.M. Group, Micro Telecommunications Computing Architecture Short Form Specification, 2006.
[8] A. Grama, Introduction to parallel computing, 2nd ed.. ed., Harlow, England ; New York : Addison-Wesley, Harlow, England ; New York, 2003.
[9] PRODRIVE, AMC-TK2: ARM and DSP AMC, http://www.ntp.org/ntpfaq/NTP-s-algo.htm, 2016.
[10] T. Instruments, Multicore DSP+ARM KeyStone II System-on-Chip (SoC), 2013.
[11] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008, DOI 10.1109/IEEESTD.2008.4610935(2008) 1-70.
[12] A. Devices, AD9361 Data Sheet Rev. E, 2014.
[13] S.H. Fuller, RapidIO : the embedded system interconnect, Chichester, England ; Hoboken, NJ : John Wiley & Sons, Chichester, England ; Hoboken, NJ, 2005.
[14] T. Barry Wood, Backplane tutorial: RapidIO, PCIe and Ethernet, EE Times, EE Times, 2009.
[15] D. Bueno, C. Conger, A.D. George, I. Troxel, A. Leko, RapidIO for radar processing in advanced space systems, ACM Trans. Embed. Comput. Syst, 7 (2007).
[16] N.P. (R&D), What is NTP?, http://www.ntp.org/ntpfaq/NTP-s-def.htm.
[17] N.P. (R&D), How accurate will my Clock be?, http://www.ntp.org/ntpfaq/NTP-s-algo.htm.
[18] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems (1588-2008), USA: IEEE, USA, 2008.
[19] D.M. Anand, J.G. Fletcher, Y. Li-Baboud, J. Moyne, A practical implementation of distributed system control over an asynchronous Ethernet network using time stamped data, 2010, pp. 515-520.
[20] S.F. Reddaway, P. Bruno, P. Rogina, R. Pancoast, Ultra-high performance, low-power, data parallel radar implementations, Ieee Aerospace and Electronic Systems Magazine, 21 (2006) 3-7.
[21] H.T. Kung, C.E. leiserson, Systolic Arrays (for VLSI), Sparse Matrix Proceedings 1978, SIAM, Philadelphia, 1979, pp. 256-282.

[22] M.A. Richards, J.A. Scheer, W.A. Holm, Priniciples of Modern Radar, Scitech Publishing, Edison, NJ, 2010.
[23] A. Klilou, S. Belkouch, P. Elleaume, P. Le Gall, F. Bourzeix, M.M.R. Hassani, Real-time parallel implementation of Pulse-Doppler radar signal processing chain on a massively parallel machine based on multi-core DSP and Serial RapidIO interconnect, EURASIP Journal on Advances in Signal Processing, 2014 (2014) 1-22.
[24] T. Instruments, Enhanced Direct Memory Access 3 (EDMA3) for KeyStone Devices User's Guide (Rev. B), 2015 May.