Exploiting FPGAs and GPUs for Electromagnetics Applications: Interferometric Imaging in Random Media Case Study

E. El-Araby, O. Kilic, and V. Dang

Department of Electrical Engineering and Computer Sciences
The Catholic University of America, Washington, DC 20064, U.S.A.
aly@cua.edu, kilic@cua.edu, and 13dang@cardinalmail.cua.edu

Abstract **─** There is a growing need for reliable and efficient numerical methods for electromagnetic applications. This is important for addressing the complex designs with fine features on electrically large platforms. As designs become more complex, a good prediction of overall system performance becomes essential for cost reduction especially in the conceptualization stage. Researchers have attempted to address this issue by developing hybrid methods based on asymptotic techniques that can avoid the numerical inefficiency while maintaining high degrees of accuracy. Another approach is to implement fast computational methods that utilize parallel computing platforms. This paper focuses on the latter; i.e. by investigating the use of field programmable gate arrays (FPGA) and general purpose graphics processing units (GPGPU) as coprocessors to parallelize numerically challenging problems. The weaknesses and strengths of both platforms will be investigated in the context of their ease of use, efficiency, and potential for accelerated computations.

Index Terms **─** FPGA, GPU, hardware accelerated computing, high performance computing, imaging, interferometry, numerical methods, random media.

I. INTRODUCTION

In the field of electromagnetic modeling, the complex designs for engineered materials coupled with performance analysis of radio frequency components integrated within their natural environment drive the need for highly efficient numerical techniques. This cannot be achieved by conventional computer systems, but rather through using the so-called high performance computing (HPC) systems. HPC systems often utilize an integrated package of multiprocessors, multicore processors, and specialized hardware enabling rapid computations by exploiting the parallelism of these hardware platforms. Various configurations and platforms exist including clusters that utilize multiple CPUs (e.g., Cray), PCs supported by GPGPU (e.g., NVIDIA, AMD) and FPGA (e.g., Xilinx, Altera) based systems. In the current state of the art, both GPGPU and FPGA systems use these hardware as coprocessors. Such configuration and usage is typically referred to as hardware acceleration.

GPGPUs have been increasingly utilized for accelerated computing in numerous fields as they are readily integrated in PCs. Their main advantages are the high memory bandwidth as well as the availability of multiple vendors developing commercial tools that enable high level language support. FPGAs, on the other hand, are highly customizable and reconfigurable chips which can be optimally configured for a specific application.

This paper investigates these hardware acceleration platforms for computational electromagnetics applications, and is an extension of the work reported earlier in [1]. The application used for this investigation is the interferometric imaging of objects behind random media. A detailed discussion on the different architectures and programming environments for FPGA and GPGPU based systems is presented in Section II. Interferometric imaging of targets behind random media is chosen as the application. The details of the interferometric imaging are introduced in Section III. The implementation approaches on both platforms are given in Section IV. Performance analysis and metrics for evaluation are presented in Section V. The experimental

Submitted On: Oct. 12, 2011
Accepted On: Jan. 21, 2012

results are provided in Section VI, followed by the conclusions in Section VII.

II. HARDWARE ACCELERATION

Hardware acceleration is the use of hardware to enable parallel processing for higher computation speed than is possible in software running on the general purpose CPU. Examples of hardware acceleration are systems that include field programmable gate arrays (FPGA) and/or general purpose graphics processing units (GPGPU).

A. FPGA-based systems

The evolution of hardware acceleration based on reconfigurable computers (RC), such as FPGAs, has been progressing along two orthogonal technology-characterizing paths, namely performance and flexibility. RCs evolved from originally being discrete components used mainly as glue-logic devices in larger systems to accelerator boards and recently to parallel reconfigurable supercomputers often referred to as high-performance reconfigurable computers (HPRCs). Examples of such supercomputers are the SRC-7 and SRC-6 [2], the SGI Altix/RASC [3] and the Cray $XT5_h$ and Cray XD1 [4].

Reconfigurable computing (RC) architectures

A reconfigurable computer system typically consists of microprocessor and reconfigurable processor sub-systems closely coupled with each other through a common interface. The microprocessor sub-system includes all major components of a traditional computer system such as general purpose microprocessors, microprocessor memories, and I/O interfaces. On the other hand, a reconfigurable processor sub-system consists of one or more FPGAs, FPGA memories, and an I/O interface. A generalized architecture of a reconfigurable computer is shown in Fig. 1. The functionality of each component within the system is influenced by its interconnection topology. Within a reconfigurable processor sub-system, every FPGA can be connected to every other FPGA, or FPGAs can be grouped into clusters. FPGA memories can be shared by a group of FPGAs or each can be dedicated to a single FPGA. Additional hardware, such as a cross-bar switch, might be necessary to make connections as flexible as possible.

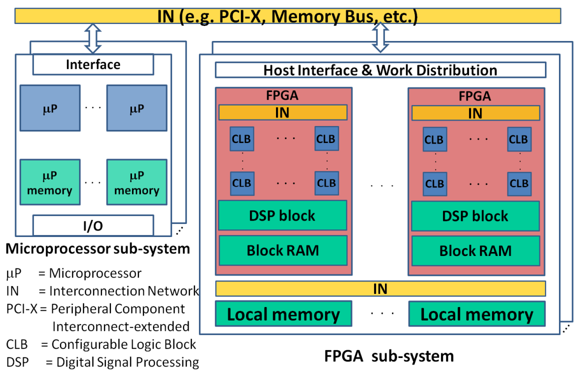
IN (e.g. PCI-X, Memory Bus, etc.)
Interface
µP
µP
µP memory
µP memory
I/O
Microprocessor sub-system
µP = Microprocessor
IN = Interconnection Network
PCI-X = Peripheral Component Interconnect-extended
CLB = Configurable Logic Block
DSP = Digital Signal Processing
Host Interface & Work Distribution
FPGA
IN
CLB
DSP block
Block RAM
IN
Local memory
Local memory
FPGA sub-system

Fig. 1. General architecture of RC systems.

RC programming models

Application development on RC systems typically requires software and hardware programming expertise for which design paradigms and tools have been traditionally separate [1]. These products aim to abstract underlying hardware design details and streamline the disparate design flows [5]. They often tradeoff performance for programmability [6]. Dataflow design tools, based on the graphical user interface, e.g., DSPLogic, seem to offer an interesting compromise between high-level languages (HLLs) and hardware description languages (HDLs), e.g. VHDL and/or Verilog. They allow a tradeoff between a shorter development time and a performance overhead imposed by HLLs [7]. Streamlining hardware description using HLLs typically used in software programming, or at least using dataflow languages, is a major and distinctive feature of high performance RCs that potentially allows domain scientists to develop entire applications without relying on hardware designers. However, an HLL compiler for RCs must combine the capabilities of tools for traditional microprocessor compilation and tools for computer-aided design with FPGAs. It must also extend these two separate set of tools with capabilities for mutual synchronization and data transfer between microprocessors and reconfigurable processor sub-systems [8].

B. GPGPU-based systems

Due to their powerful floating-point computational capabilities and massively parallel processor architecture, GPUs are increasingly being used as application accelerators in the high-performance computing arena. Thus, a wide range of HPC systems now incorporate GPUs as hardware accelerators including systems ranging from clusters of compute nodes to parallel supercomputer systems. Several examples of GPU-based computer clusters may be found in academia, such as [9]. Latest offerings from supercomputer vendors have begun to include GPUs in the compute blades of their parallel machines; examples include the SGI Altix UV and the recently announced plans from Cray to include NVIDIA GPUs into the Cray XE6 supercomputer.

GPGPU architecture

Similar to an RC system, a GPGPU computer system typically consists of a traditional microprocessor sub-system and a GPU sub-system closely coupled with each other through a common interface. A GPU sub-system consists of a multitude (texture/cluster) of processors often referred to as streaming multiprocessor (SM) and GPU local memory. Each SM contains a number of processor cores usually called streaming processors (SP). A generalized architecture of a GPGPU computer is shown in Fig. 2. For example, the Fermi architecture, which is one of the most recent GPU products by NVIDIA, is composed of 16 streaming multiprocessors (SMs) each of which consists of 32 streaming processor (SP) cores.

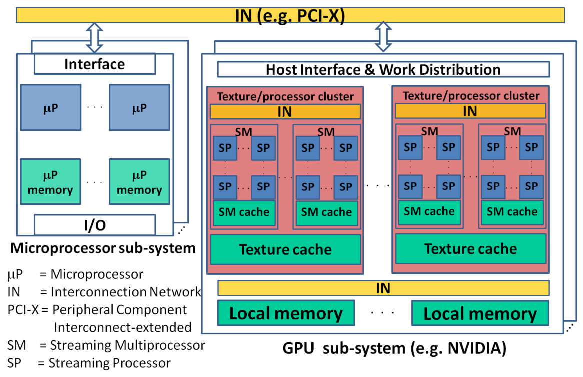
IN (e.g. PCI-X)
Interface
µP
µP
µP memory
µP memory
I/O
Microprocessor sub-system
Host Interface & Work Distribution
Texture/processor cluster
IN
SM
SP
SM cache
Texture cache
IN
Local memory
Local memory
GPU sub-system (e.g. NVIDIA)
µP = Microprocessor
IN = Interconnection Network
PCI-X = Peripheral Component Interconnect-extended
SM = Streaming Multiprocessor
SP = Streaming Processor

Fig. 2. General architecture of GPGPU-based systems.

GPGPU programming models

Among the different parallel programming approaches for GPGPU platforms, the most commonly followed programming style is the single program multiple data (SPMD) model [10]. Under the SPMD scenario, multiple processes execute the same program on different CPU cores, simultaneously operating on different data sets in parallel. By allowing autonomous execution of processes at independent points of the same program, SPMD serves as a convenient yet powerful approach for efficiently making use of the available hardware parallelism.

Two widely used GPU programming models, i.e., CUDA (by NVIDIA) and OpenCL (by Khronos Working Group) are similar and follow SPMD flow by executing data-parallel kernel functions within the GPU. Both models also provide abstractions of thread (basic unit for parallel execution) group hierarchy and shared memory hierarchy. In terms of thread group hierarchy, both provide three hierarchy levels: grid, block, and thread. GPU kernels are launched per grid and a grid is composed of a number of blocks, which have the access to the global device memory. Each block consists of a group of threads, which are executed concurrently and share the accesses to the on-chip shared memory. Each thread is a very lightweight execution of the kernel function. From programming perspective, the programmer needs to write the kernel program for one thread and decides the total number of threads to be executed on the GPU device while dividing the threads into blocks based on the data-sharing pattern, memory sharing, and architectural considerations.

III. INTERFEROMETRIC IMAGING IN RANDOM MEDIA

The earliest applications of interferometric measurements have been reported in the field of radio astronomy [11,12]. The main advantage of interferometry is the higher resolution achieved by the use of multiple antennas instead of using an equivalent larger antenna.

An interferometric image is created using the complex correlations of intensities obtained from all possible pair combinations in a detector array [13,14]. This is expressed as follows:

$$\sigma_E(\zeta,\eta)=\sum_{i=1}^{N(N-1)/2} A_i\cos(\Delta\phi_i+k(\mu_i\zeta+\nu_i\eta)), \quad (1)$$

where σ_E denotes the time average intensity of the source, ζ and η correspond to the coordinates of each pixel in the image, $\mu_i = x_n - x_m$ and $\nu_i = y_n - y_m$ are the x and y baselines provided by detector pairs (x_m,y_m) and (x_n,y_n), k is the wave number, and N is the number of detectors in the array. The summation is evaluated over all possible combinations of detector pairs. A_i and $\Delta\phi_i$ correspond to the amplitude and phase terms of the correlated fields for the mn^{th} pair such that, $A_i = |E_m E_n^*|$ and $\Delta\phi_i = \phi_m - \phi_n$.

A. Target behind random media

Interferometric imaging has recently been applied to targets behind random media [15], and implemented on GPGPU platform [16]. A summary of these research outcomes are presented in this section. Furthermore, the same algorithm is implemented on an FPGA platform to gain insight into the differences in implementation on GPGPU versus FPGA platforms. We report on how the different computer architectures are utilized in the implementation, and their performance levels in terms of speed and efficiency.

Our model for interferometric imaging of targets behind random media consists of three major building blocks: (i) field calculation at the detector array (includes scattered field from the target and random medium), (ii) image construction, and (iii) image plotting. The field calculations include a direct scattering term from the target, a direct scattering term from the media as well as an indirect term scattered from the target to the medium before arriving at the detector array, see Fig. 3. The scattering field calculations from the random media uses the distorted Born approximation following the work reported earlier, [17]. Attenuation effects are accounted for as the fields travel through the random medium.

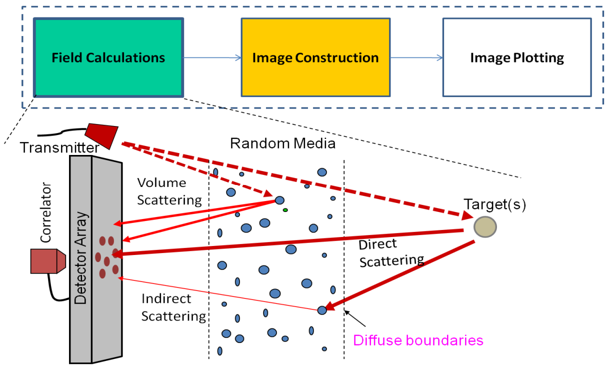

Fig. 3. Computational steps of the model.

B. Parallelization

The most time consuming computation in the model is the image construction, which consists of the summation of correlated fields for each pair of detectors as given in (1). Since the intensity for each pixel can be calculated independently, this part of the algorithm can benefit from parallel implementation. Therefore, the image construction was selected as the candidate for hardware implementation, while the rest of the algorithm is carried out on the CPU.

IV. HARDWARE IMPLEMENTATION

For comparison purposes, we start with a highly productive programming environment for both GPGPU and FPGA implementations to benefit from a fast learning curve. For the FPGA implementation, we employed the SRC-6 system and its proprietary Carte programming environment. For the GPGPU implementation, we utilized Jacket version 1.5.0 by AccelerEyes on an NVIDIA C1060 based system. All field calculations and image plotting are performed on the CPU. The image construction stage, which is implemented on hardware, is divided into four stages: (i) setup of the hardware, (ii) transfer of input parameters to the hardware, (iii) parallelized calculations, and (iv) transfer of results back to the CPU. The specifics of the implementation on these two systems are discussed in the following sub-sections.

A. Implementation on FPGA

The FPGA platform employed for the implementation is the SRC-6 scalable system which is a cluster-based system. The SRC-6 system includes two MAP (multi-adaptive processor) reconfigurable boards and each consists of two FPGAs, which can all be programmed simultaneously in one application. Each MAP has 24 MB of memory and the processing speed of FPGAs is 100 MHz. SRC's proprietary Carte-C programming environment is used in the code development [2].

In the FPGA implementation, the image is divided into two regions and run in parallel on a single FPGA. The computations and the summation given in (1) are pipelined such that the accumulation pipeline can start before all terms are computed. SRC's Carte development environment provides a relatively easy interface to implement code on the FPGA, while allowing the programmer some control over the data streaming and parallelization.

B. Implementation on GPGPU

The GPGPU platform used in the implementation is a single workstation which consists of a 16-core 2.67 GHz Intel Xeon CPU with four NVIDIA Tesla C1060 graphics processors with a total 16 GB of memory and runs at 1.3 GHz. The workstation operates on Microsoft Windows 7 Professional, and MATLAB version 7.7.0.471 (R2008b) is utilized along with Jacket version 1.5.0 by AccelerEyes. This version of Jacket uses NVIDIA's CUDA version 3.1. The architecture of the system is similar to the general architecture shown earlier in Fig. 2.

Jacket programming environment from Accelereyes is used in the implementation. This package can be easily integrated with MATLAB and provides a MATLAB-like environment to enable a fairly easy learning curve. However, this comes at the expense of limited control over how the data streaming and parallelization are implemented, resulting in reduced efficiency as compared to that of FPGAs.

V. PERFORMANCE ANALYSIS AND METRICS FOR EVALUATION

In our analysis, we assumed the fixed-time model (Gustafson's Law) [18] and used larger computational workloads (larger data/image size) while maintaining the same performance on larger configurations. Larger configurations utilize more processing units (FPGAs or GPUs) which results in increasing the overall system utilization.

The image construction is composed of 5 stages as depicted in Fig. 4. The algorithm first sets up the parallel environment, then transfers data from CPU memory space to accelerator memory space, processes data using accelerator(s), transfers the image data back to the CPU and finally releases the allocated hardware resources. The associated execution times for each of these stages are defined as shown in Fig. 4.

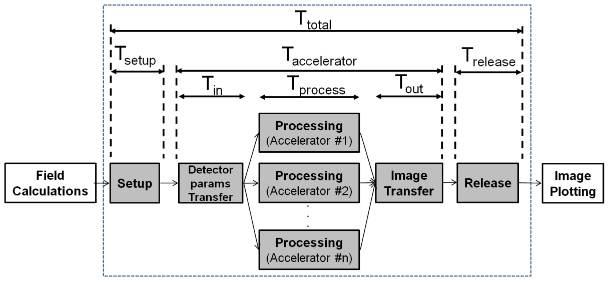

Fig. 4. Execution model and performance metrics.

Execution scenarios

Two scenarios were considered for hardware implementation, namely no-streaming and streaming scenarios. In the no-streaming scenario, operations are performed in a non-overlapped execution while in the streaming scenario operations are allowed to overlap, as shown in Figs. 5 and 6, respectively. These scenarios affect the total execution time as given in (2) and (3).

$$T_{total}^{no-streaming} = T_{setup} + T_{in} + T_{process} + T_{out} + T_{release}, \quad (2)$$

$$T_{total}^{streaming} = T_{setup} + T_{in} + \max(T_{process}, T_{out}) + T_{release}. \quad (3)$$

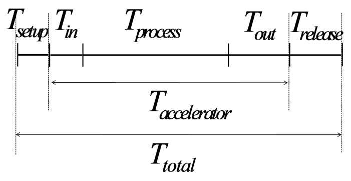

Fig. 5. No-streaming (non-overlapped execution).

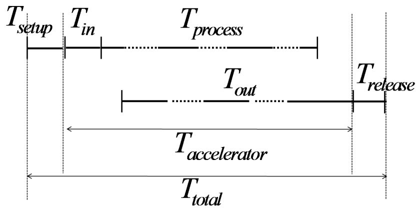

Fig. 6. Streaming (overlapped execution).

Metrics

Three metrics were investigated for performance comparison between the two platforms: (i) speedup, (ii) efficiency, and (iii) scalability.

The speedup is defined as the performance gain by using multiple hardware processing units (i.e. GPU, FPGA) in reference to a single CPU. This can be expressed as follows:

$$S(N_{PU}) = \frac{T_{total}^{CPU}(1, N_{PU}D_0)}{T_{total}^{PU}(N_{PU}, N_{PU}D_0)}, \quad (4)$$

where D_0 is the image size for a single processing unit, and N_{PU} is the number of processing units. $T_{total}^{CPU}(1, N_{PU}D_0)$ is the total execution time of a single CPU with image size of $N_{PU}D_0$, and $T_{total}^{PU}(N_{PU}, N_{PU}D_0)$ is the total execution time of multiple processing units with image size of $N_{PU}D_0$.

The hardware efficiency compares the execution time measured through experiments to the expected/theoretical performance. This is expressed as:

$$E(N_{PU}, N_{PU} D_0) = \frac{T_{total}^{th}(1, D_0)}{T_{total}^{meas}(N_{PU}, N_{PU} D_0)} (\%), \qquad (5)$$

where $T_{total}^{th}(1, D_0)$ is the theoretical total execution time of a single processing unit with image size D_0, and $T_{total}^{meas}(N_{PU}, N_{PU} D_0)$ is the measured total execution time of multiple processing units with image size of $N_{PU} D_0$. The theoretical execution time, $T_{total}^{th}(1, D_0)$ is calculated using (2) or (3) depending on the implemented execution scenario. Each term in (2) and (3) is calculated as discussed below.

The setup and release times represent the system overhead associated with setting up and releasing multiple processing units and therefore they are both equal to zero for a single GPU. However, the setup time for a single FPGA is non-zero, i.e. 65 ms, due to configuring the FPGA.

The input transfer time is the ratio of the total amount of input transfer data, D_{in}, to the bandwidth, B_{in}, for the input data transfer from the CPU's memory to accelerator's memory. The input data size, D_{in}, is calculated as:

$$D_{in} = N_{det-pairs} . N_{param}^{detector} . D_{bytes}, \qquad (6)$$

where $N_{det\text{-}pairs}$ is the number of detector pairs, D_{bytes} is the number of precision bytes, and $N_{param}^{detetor}$ is the number of detector parameters. The input bandwidth values are as shown in Table 1.

Table 1: Bandwidth parameters

(MB/s)	B_{in}	$B_{process}$	B_{out}
GPU	2,169	61,360	1,151
FPGA	1,415	4,800	1,260

The processing time is the ratio of the total amount of data for processing, $D_{process}$ to the processing throughput, $B_{process}$. $B_{process}$ is given in Table 1 and $D_{process}$ is calculated by equation (7):

$$D_{process} = N_{pixels} . N_{det-pairs} . (N_{param}^{detector} + N_{param}^{pixel}) . D_{bytes}. \qquad (7)$$

The output transfer time is the ratio of the total amount of output transfer data, D_{out} to the bandwidth for the output data transfer, B_{out}, from the accelerator's memory to the CPU's memory. The output transfer data size is calculated by

$$D_{out} = N_{pixels} . D_{bytes}, \qquad (8)$$

where N_{pixels} is the number of pixels in the image, D_{bytes} is the number of precision bytes as before, and the bandwidth, B_{out} is given in Table 1.

Finally, the scalability factor, Ω, is defined as the normalized speedup. In other words, it is the performance achieved by multiple processing units as compared to the performance achieved by a single processing unit as given by equation (9).

$$\Omega(N_{PU}) = \frac{S(N_{PU})}{S(1)}. \qquad (9)$$

VI. EXPERIMENTAL RESULTS

The image size for a single processing unit (i.e. GPU or FPGA) is chosen as D_0 = 250,000 pixels for comparison purposes. The implementation for both GPU and FPGA were parallelized using up to four processing units in each system. Therefore, as the number of processors was increased, proportionally more pixels were created in the image (following Gustafson's model). A comparison is provided in Fig. 7 in terms of the end-to-end computation time for each system, as well as the hardware-only time. Figure 7 also shows the speedup of total execution time of GPUs/FPGAs over the total execution time of a single CPU which are put in parentheses next to their corresponding execution times. It is observed for the unit data size implemented on a single processing unit, GPU outperforms the FPGA (speedup of 71 versus 12). As the image size increases, increasing the number of FPGAs compensates for added computational workload, and the total time remains constant. However, the GPU system shows signs of decreased efficiency with increased workload (image size). This can be better observed if one considers the ratio of the time it took to construct the image on the FPGA system versus the GPU system. It is observed that for the single accelerator case, the GPU system is approximately six times faster than the FPGA. As the number of accelerators is increased (proportionally with the pixels generated), the ratio drops to roughly two. This is due to the higher efficiency achieved by the FPGA implementation as well as the faster decreasing efficiency of the GPU implementation. The decreasing efficiency of the GPU is due to the overhead associated with the setup and merging of the results before transferring the output back to the CPU . The GPU requires more resources to be

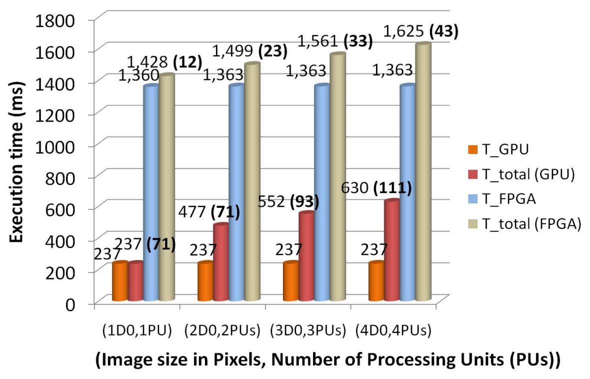

Fig. 7. Total execution time.

allocated and becomes less efficient.

The system efficiency was calculated by taking the ratio of the measured time to the theoretically expected time for each system to complete the calculations. The theoretical time calculations are based on the system parameters such as the bandwidth, processing speed, etc. The efficiencies for both implementations are depicted in Fig. 8, demonstrating that despite being slower than the GPUs, FPGA implementations show higher efficiency.

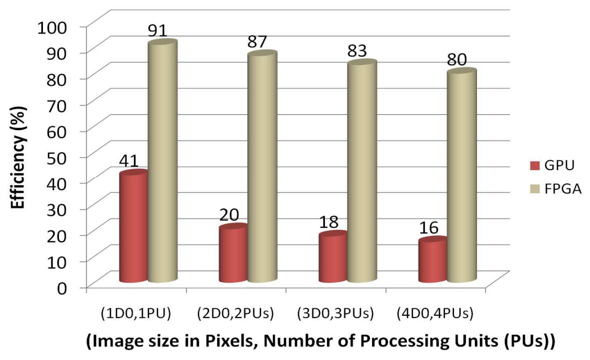

Fig. 8. Efficiency of FPGA vs. GPU.

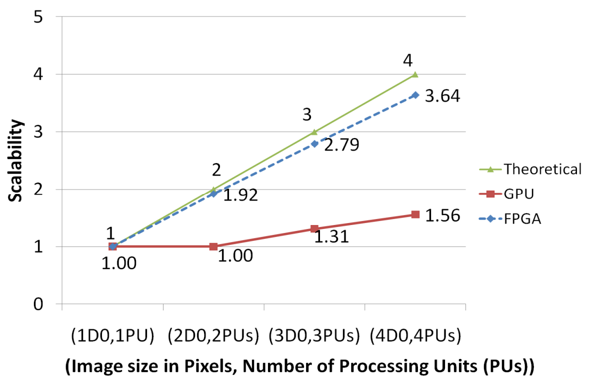

Fig. 9. Scalability of algorithm on FPGA vs. GPU.

In order to investigate the scalability, we compare the speedup scale with increasing image size as depicted in Fig. 9. It is observed that the FPGA computations when compared to GPU computations scaled more closely to the theoretical linear expectations.

VII. CONCLUSIONS

While the comparison of performance in terms of execution time shows that the GPU based implementation performs better than the FPGA, it should be noted that the FPGA system has significantly more limited resources in terms of clock speed (100 MHz vs. 1.3 GHz) and on board memory (48 MB total vs. 16 GB total). From that perspective, the FPGA implementation proves to be significantly more efficient than the GPU implementation. This is shown in the comparison of the system efficiencies, where the FPGA implementation achieved about 90% efficiency as opposed to the 40% observed for the GPU implementation. It is worthwhile noting that the SRC Carte programming environment allows the user flexibility in controlling the data utilization in terms of pipelining and parallelization, whereas with Jacket environment the user is oblivious to how the GPU is controlled. This comes at the expense of ease of use, as Jacket is very much like MATLAB and very easy to use, while the learning curve in Carte is steeper. The tradeoff between productivity versus efficiency is one of the key features for hardware accelerated computing on these platforms. While FPGAs can be programmed to perform a specific task in a highly efficient way, they currently lack the wide commercial support enjoyed by GPGPUs. Consequently, they require a good knowledge of hardware and ability to program in hardware languages such as VHDL and/or Verilog to program them most efficiently.

ACKNOWLEDGMENT

The authors would like to thank Prof. Tarek El-Ghazawi, the founding director of the George Washington University (GWU) High Performance Computing Laboratory (HPCL), for making available to us the SRC-6 system to conduct the experimental work of this research. We would also like to thank the HPCL team, in particular Olivier Serres, for their help and technical support.

REFERENCES

[1] O. Kilic, E. El-Araby, and V. Dang, "Hardware Accelerated Computing for Electromagnetics Applications," *International Workshop on Computational Electromagnetics (CEM'11)*, Izmir, Turkey, August 2011.

[2] SRC Computers, Inc., "SRC Carte^{TM} C Programming Environment v2.2," *(SRC-007-18)*, Aug. 2006.

[3] Silicon Graphics Inc., "Reconfigurable Application-Specific Computing User's Guide," *(007-4718-005)*, Jan. 2007.

[4] Cray Inc., "Cray XD1^{TM} FPGA Development," *(S-6400-14)*, 2006.

[5] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. B. Newby, "Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study," *III Southern Conference on Programmable Logic (SPL2007)*, Mar del Plata, Argentina, February, 2007.

[6] O. Kilic, "FPGA Accelerated Phased Array Design using the Ant Colony Optimization," *Applied Comp. Electromag. Soc. Journal*, Vol. 20, No. 1, pp. 23-30, February 2010.

[7] E. El-Araby, P. Nosum, and T. El-Ghazawi, "Productivity of High-Level Languages on Reconfigurable Computers: An HPC Perspective," *IEEE International Conference on Field-Programmable Technology (FPT 2007)*, Japan, December, 2007.

[8] El-Araby, S. G. Merchant, and T. El-Ghazawi, "A Framework for Evaluating High-Level Design Methodologies for High-Performance Re-configurable Computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, pp. 33-45, Jan. 2011.

[9] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu, "GPU Clusters for High-Performance Computing," *Proc. Workshop on Parallel Programming on Accelerator Clusters* , 2009.

[10] F. Darema, "The SPMD Model: Past, Present and Future," *Proc. 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing interface*, Lecture Notes In Computer Science, vol. 2131/2001, pp. 1, Sept. 2001.

[11] M. Ryle, "A New Radio Interferometer and Its Applications to the Observation of Weak Radio Stars," *Proc. Roy. Soc. London Ser. A*, vol. 211, pp. 351-375, 1952.

[12] R. N. Bracewell, "Radio Interferometry of Discrete Sources," *Proc. IRE*, vol. 46, pp. 97-105, 1958.

[13] J. F. Frederici, D. Gray, B. Schulkin, F. Huang, H. Altan, R. Barat, and D. Zimdars, "Terahertz Imaging Using an Interferometric Array," *Appl. Phys. Lett.*, vol. 83, p. 2477, 2004.

[14] K. P. Walsh, B. Schulkin, D. Gary, J. F. Federici, R. Barat, and D. Zimdars, "Terahertz Near-Field Interferometric and Synthetic Aperture Imaging," *Proc. SPIE*, vol. 5411, p. 9, 2003.

[15] O. Kilic and A. Smith, "Imaging Through Random Media," *European Conference on Antennas and Propagation* (*EuCAP)*, Rome, Italy, April 2011.

[16] O. Kilic, A. Smith, E. El-Araby, and V. Dang, "Interferometric Imaging Through Random Media using GPU," *The Applied Computational Electromagnetics Society (ACES)*, Williamsburg, VA, USA, April 2011.

[17] O. Kilic and R. H. Lang, "Scattering of a Pulsed Beam by a Random Medium Over Ground," *J. of Electromagnetic Waves and Appl.*, vol. 15, no. 4, pp. 481-516, 2001.

[18] K. Hwang, and Z. Xu, "Scalable Parallel Computing: Technology, Architecture, Programming," McGraw-Hill, 1998.

Dr. Esam El-Araby received his B.Sc. degree in Electronics and Telecommunications Engineering and his M.Sc. degree in Automatic Control and Computer Engineering from Assiut University, Egypt, in 1991 and 1997, respectively. He received his M.Sc. and Ph.D. degrees in Computer Engineering from the George Washington University (GWU), USA, in 2005 and 2010, respectively. Dr. El-Araby joined the Catholic University of America (CUA) as an Assistant Professor in the Department of Electrical Engineering and Computer Science in 2010. He is the founder and director of the Heterogeneous and Biologically-inspired Architectures (HEBA) laboratory at CUA. Prior to that, he worked at the High Performance Computing Laboratory (HPCL) at GWU as well as the NSF Center for High-Performance Reconfigurable Computing (NSF-CHREC). His research interests include computer architecture, hybrid/heterogeneous architectures, hardware acceleration, reconfigurable computing, embedded systems, evolvable hardware, performance evaluation, and digital signal/image processing and remote sensing.

Dr. Ozlem Kilic joined the Catholic University of America as an Assistant Professor in the Department of Electrical Engineering and Computer Science in 2005. Prior to that, she was an Electronics Engineer at the U.S. Army Research Laboratory, Adelphi MD where she managed Small Business Innovative Research (SBIR) Programs for the development of hybrid numerical electromagnetic tools to analyze and design electrically large structures, such as the Rotman lens. She has also designed, fabricated and tested various prototypes. Dr. Kilic has over five years of industry experience at COMSAT Laboratories as a Senior Engineer and Program Manager with specialization in satellite communications, link modeling and analysis, and modeling, design and test of phased arrays and reflector antennas for satellite communications system. Her research interests include numerical electromagnetics, antennas, wave propagation, satellite communications systems, and microwave remote sensing.

Vinh Dang received his B.Sc. degree in Electronics and Telecommunications Engineering from the Posts and Telecommunications Institute of Technology, Vietnam, in 2003. He received his M.Eng. degree in Electrical Engineering from the University of Technology, Vietnam, in 2006. He is currently a Ph.D. candidate at the Department of Electrical Engineering and Computer Science, the Catholic University of America (CUA), USA. During the period from 2008 to 2010, he was a Lecturer at the School of Electrical Engineering, International University, Vietnam. Since 2010, he has been a Research Assistant in the Department of Electrical Engineering and Computer Science at CUA. His current research interests include high performance computing, embedded systems, biomedical image processing, and remote sensing.