Efficient Software Architecture Pattern for Accelerator Based Computing

Bhogendra Rao PVRR

Abstract: Graphics Accelerators are increasingly used for general purpose high performance computing applications as they provide a low cost solution to high performance computing requirements. Intel also came out with a performance accelerator that offers a similar solution. However, the existing application software needs to be restructured to suit to the accelerator paradigm with a suitable software architecture pattern. In the present work, master-slave architecture is employed to convert CFD grid free Euler solvers in CUDA for GPGPU computing. The performance obtained using master-slave architecture for GPGPU computing is compared with that of sequential computing results.

Keywords: Software Architectures, Architecture Patterns, Parallel and Distributed Computing, GPGPU, CUDA computing.

I. INTRODUCTION

High performance computing (HPC) has been an industrial need for long time. However, the cost of HPC Systems has been prohibitively high for the users to effectively make use of the technology. General Purpose Graphics Processing Units (GPGPUs) provide a low cost solution to high performance computing. GPU is especially designed for problems that can be expressed as data-parallel computations, that is, same program is executed on many data elements (SPMD) in parallel. GPUs are designed such that they devote more transistors for data processing. As same program is executed on each data element, the requirement for sophisticated control statements in the program is low.

Data-parallel processing maps data elements to parallel processing threads. Application programs that process large data sets can use a data-parallel programming model to speed up the computations. This is also referred to as SIMT (Single Instruction Multiple Threads) architecture.

The performance of the parallel software immensely depends on the software architecture. For the benefit of software developer community various architectures patterns and design patterns have been brought out and published [1][2] in the literature. Master-slave architecture has been applied to a variety of problems in parallel computing which are discussed later in the paper. In the present work, the master-slave architecture successfully is employed in CUDA based parallel computing.

Organization of this paper is as follows: Section II gives a brief introduction to programming with CUDA, which is essential to understand the framework of CUDA computing. Various software architectures available in literature for parallel and distributed computing are discussed in Section III. This section also discusses the proposed architecture. Section IV discusses its application to implementation of a Computational Fluid Dynamics (CFD) algorithm. Section V discusses various issues involved in CUDA programming of the case study. Results are presented and discussed in section VI. The paper concludes with section VII.

II. PROGRAMMING WITH CUDA

The CUDA parallel programing model is designed to scale transparently to the increase in number of processors / cores while maintaining a low learning curve for programmers through standard programming languages such as C.

A. CUDA Programming Model

CUDA follows load-launch-read programming model, as shown in Figure 1. The initial data is loaded onto the GPU device memory. The parallel computing program called Kernel is launched. After completion of execution, the results are read from the device memory.

B. CUDA Kernels

The functionality which needs to execute in parallel on the data set needs to be declared as a `__global__` function and can be specified to the compiler. The kernel shall run in multiple threads, with each thread identified by a thread index. A number of threads are grouped to one block and number of blocks into a grid. This grid can be specified as 1D, 2D or 3D depending on the data set representation of the problem. There can be multiple kernels running simultaneously.

C. Kernel Configuration

At the time of launching the kernel, it is required to specify size of grid and size of block. Size of block should be a multiple of warp and should be less than max block size. However, the configuration depends highly on the current load of the device and the resources required by the kernel.

D. Factors that influence performance of parallel applications

The following are some of the factors that influence the

Revised Manuscript Received on March 18, 2020.
* Correspondence Author
Dr PVRR Bhogendra Rao, Ph.D,JNTU, Hyderabad.India

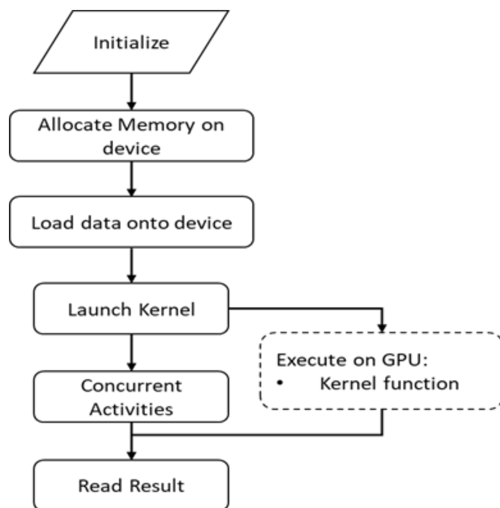


Figure 1: CUDA Programming Model

performance of parallel program.

- Load balancing
- Race conditions
- Essential sequential computations

The effective computation time per node is the time taken by the node with maximum load. Thus, the total time for computation will be effectively more than the time taken with the balanced load. Race condition is the condition when more than one thread try to access same variable and at least one of them attempts to write. In such case, it is required to synchronize the threads that are trying to access the same variable. There are certain parts of computations cannot be computed in parallel. Such essential sequential computation part should be minimized, in order to achieve higher performance of the parallel application [3].

E. Issues to be addressed while programming with CUDA

In addition to the factors discussed in previous subsection, the following are the major issues to be addressed while programming with CUDA

- Identification of suitable kernel configuration
- Coalesced access to data structure memory

The size of grid and size of block are to be correctly chosen. Otherwise the result would be erroneous. Coalesced access to memory increases the run-time performance as the data will be cached.

III. SOFTWARE ARCHITECTURES FOR PARALLEL COMPUTING

Software architectures for parallel and distributed computing applications are highly complex and are not directly applicable across applications and platforms.

A. Earlier Work

Three software architecture patterns were discussed in [1] for distributed applications. Though parallel computing is a special case of distributed computing, the patterns discussed in [1] are not suitable for parallel computing. Most commonly known software architectures for parallel and distributed computing are:

- Broker architecture

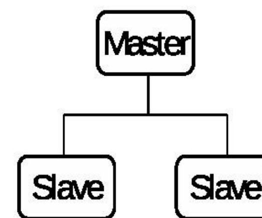


Figure 2: Software Architecture for Accelerator based Computing

- Pipes and filters architecture
- Micro kernel architecture

The Broker Pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. Platforms such as Microsoft OLE (Object Linking and Embedding) [4] and OMG's CORBA (Common Object Request Broker Architecture) [5] share a common software architecture from which the Broker pattern was abstracted.

Pipes and Filters pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems. This pattern is more often used for structuring the functional core of an application than for distribution.

The Microkernel pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core form extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinated their collaboration.

Microkernel systems employ Client-Server architecture in which clients and servers run on a top of the microkernel component. The main benefit of such systems, however, is in design for adaptation and change.

However, these architectures are not suitable for the current context, as the host program needs not only to distribute the data and to coordinate multiple devices, but also needs to control the execution.

B. Proposed Architecture

Hence the following candidate architectures were considered.

- Distributed Agent based architecture
- Socialistic architecture
- Master-slave architecture

Distributed agent based software architectures have been presented in [6] and was successfully exploited for various distributed and parallel computing applications [7][8] and [9]. Further studies were carried out on this architecture in [10] and [11].

However, the term *Agent* refers to intelligent systems [12] and hence though these architectures provide a good solution to the problem being solved, the terminology and concept are way different.

A socialistic architecture based parallel computing scheme was presented in [13] for applications based on network of workstations. In this architecture,

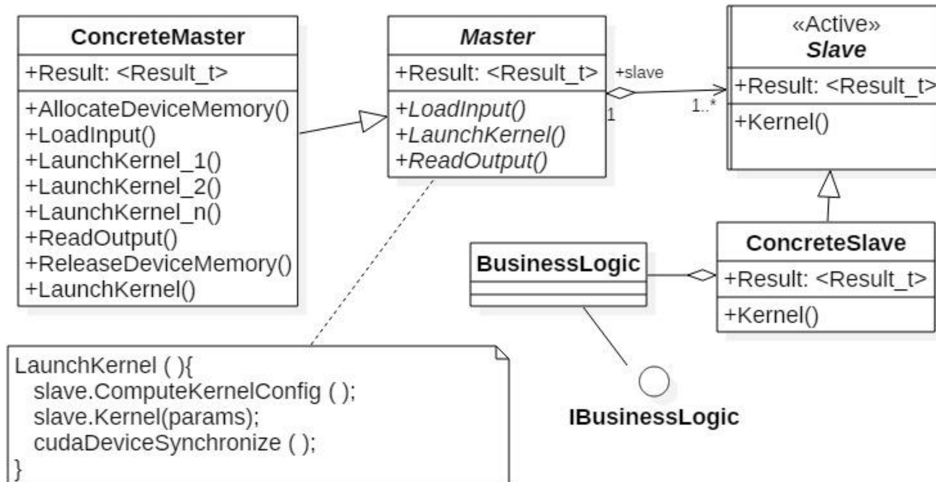


Figure 3 Master-Slave Pattern

all components perform the same functionality and exchange the intermediate results with all other components while no component controls any other component. This architecture is suitable for cluster based implementations where communication from any node to any node is possible. The Master-Slave architecture, given in Figure 2, was proposed in [14] and [15] as a candidate pattern for parallel computing applications. It was successfully exploited for deployment of application on loosely coupled multi-processor environment with proprietary message passing mechanism and was presented in [16]. This architecture was also successfully used in an airborne radar application with tightly coupled multi-processor environment [17].

C. Responsibilities of Components

The following are the responsibilities of components of Master-Slave architecture.

Class: Master	**Class:** Slave
Responsibility: 1. Read the input data from data files 2. Allocate memory on the device 3. Load the data onto device memory from host CPU memory 4. Launch the Slave(s) as kernel(s) 5. Synchronize data 6. Read the results from device memory to the host CPU	**Responsibility:** 7. Implements business logic
Collaborators: Slave	**Collaborators:** --

Often, there will be multiple slaves components and the Master component needs to spawn or launch the slaves either in sequence or in parallel depending on the application. In this particular application, the slave components are kernel modules that run on the GPGPU and perform the required CFD computations. The association among the components is given Figure 3.

IV. CONVERSION OF GRID-FREE EULER SOLVER USING CUDA PROGRAMMING

In the present work, q-LSKUM grid-free Euler Solver [18] is converted to CUDA programming environment to run on NVidia Tesla K20-m GPGPUs to reduce the time required for each simulation. The performance of grid free Euler solvers with CUDA is presented here by applying them for 2-D and 3-D problems.

A. Hardware Configuration

The software was tested on a GPGPU based high performance computing system with following configuration.

- Number of GPU cores: 2688
- Memory size: 6 GB
- clock speed: 0.732 GHz
- Memory Bandwidth: 250 Gbps
- Performance (SP): 3.95 TFLOPS
- Performance (DP): 1.31 TFLOPS
- Max Power Usage: 235 W

B. CUDA implementation of Grid-free 2D Euler Solver

The parallel implementation model of q-LSKUM 2-D grid-free Euler solver is given in fig. 4. The main data structure of 2D grid-free Euler solver is an array of structures, each structure containing node attributes and an array of pointers to neighboring nodes.

During the initialization process, the initial data and node structure are read from a file followed by memory allocation on the GPU device. Before the kernel is launched on the GPU, the array of structures is copied onto the device memory. After successful copying of the required data, the kernel is launched with the most appropriate configuration computed based on the present occupancy of the GPU.

For a grid-free Euler solver, each thread has to access various nodes that are not contiguous in the array.

In addition, this calls for synchronization among threads of different blocks. Hence, the code has to be divided into multiple kernels, based on the access to the common device memory containing the node array.

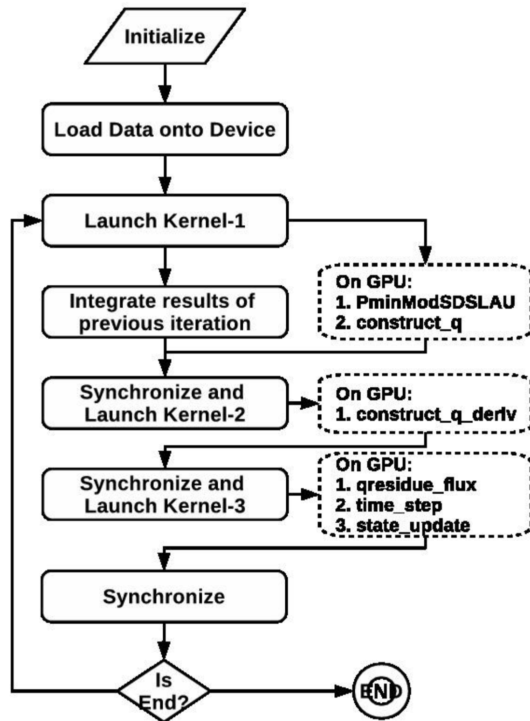


Figure 4: CUDA Programming Model

These kernels are launched sequentially one after the other, ensuring data synchronization between successive kernel launches. The first kernel performs `PminModSDSLAU()` and `connstruct_q()` functions, the second kernel performs `construct_q_deriv()` function and the third kernel performs `qresidue_flux()`, `time_step()` and `state_update()`.

In the present application, some part of the code has to be essentially run sequentially on the host leading to overall performance degradation. However, the heterogeneous computing capability of CUDA has been successfully employed to improve the performance of the overall application. The essential sequential part, the aerodynamic force integration, is the last phase of the computation in each iteration. This essential sequential part is executed on host in parallel to the following iteration on the GPU device. The integrate function, which is the essential sequential part, is executed on the CPU in parallel with Kernel-1 and Kernel-2.

C.CUDA Implementation of Grid-free 3D Euler solver

The philosophy of CUDA program model for 3D grid-free Euler Flow Solver is same as that of 2-D grid-free Euler Flow Solver given in fig. 4. In the implementation of CUDA program model for 3D grid free Euler solver, one-time computation of least square coefficients is launched as a CUDA kernel before entering into iteration loop. Generic wing-store separation problem with coarse grid with 42,664 points has been considered as test case to check the efficiency of the CUDA program.

V.IMPLEMENTATION ISSUES

The issues discussed in sections 2 have been addressed and the code was successfully implemented with CUDA. These issues are addressed as follows.

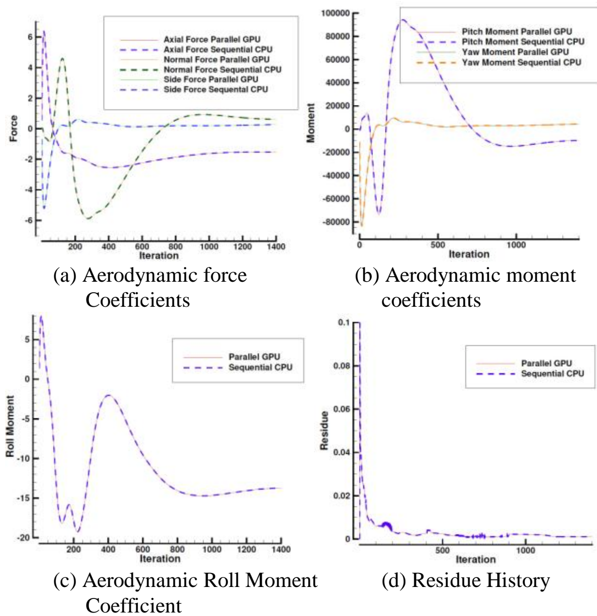
Listing 1. Computation of Kernel Configuration

```
int gridSize;
int blockSize;
int MinGridSize;

cudaOccupancyMaxPotentialBlockSize (
                  &minGridSize,
                  &blockSize,
                  (void *)KernelFunction,
                  0,
                  MaxNodes);

gridSize = (MaxNodes + blockSize - 1 ) /
                  blockSize;
```

1) The kernel configuration is computed based on the current load on the GPGPU. Exact configuration, that can successfully launch the kernel, was computed from the results returned by `cudaOccupancyMaxPotentialBlockSize ( )` function [19][20]. The code for performing this functionality is given in listing 1.
2) Race conditions were avoided by identifying sections of the code where global shared memory was simultaneously accessed by multiple threads with at least one of the accesses is for write operation. The code was divided into multiple Slaves (kernels) at those sections of code. However, if the threads are trying to access the variable from other blocks, they needed to be synchronized using `cudaSynchronizeDevice( )`. This function can be called from the host code only.
3) Integration of aerodynamic forces is the essential sequential computation in the present problem. Here, the heterogeneous computing capability of GPGPU has been effectively utilized. The aerodynamic forces of ith iteration are integrated on the CPU, while the $(i+1)^{th}$ iteration is executed on the GPGPU. Integration of aerodynamic forces corresponding to the last iteration is performed sequentially after all iterations are completed.
4) The problem being a grid-free Euler Flow solvers, the number of neighboring points for different points in the grid are different and the amount of computation is proportional to the number of neighboring points.
5) Despite the recommendation in [21], problem being grid-free, coalesced memory access could not be achieved.

VI. RESULTS

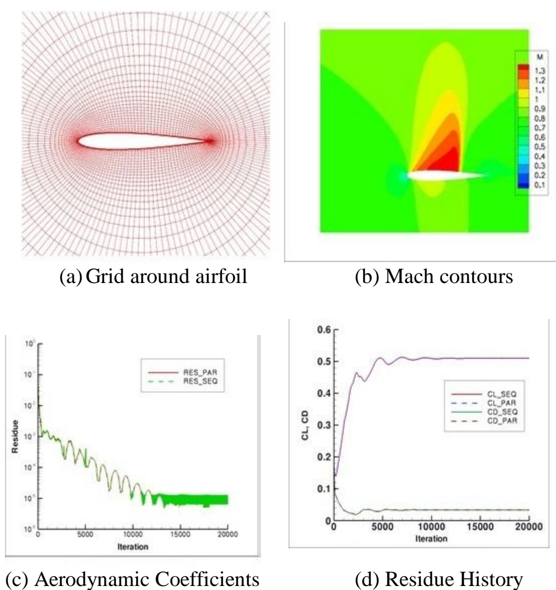
CUDA implementation of grid free Euler solver was carried out on GPGPU. The grid free solver is applied to 2-D and 3-D test cases to verify the repeatability of the results and to compare with sequential version on CPU.



(a) Aerodynamic force Coefficients (b) Aerodynamic moment coefficients

(c) Aerodynamic Roll Moment Coefficient (d) Residue History

Figure 5. Results of 3D Euler Flow Solver using GPGPU



(a) Grid around airfoil (b) Mach contours

(c) Aerodynamic Coefficients (d) Residue History

Figure 6. Results of 2D Euler Flow Solver using GPGPU

A. Results of 2D and 3D Euler flow solver on GPGPU

Similar work was carried-out using GPGPU for 2D meshless method for solving compressible flow problems and was presented in [21]. In the literature, for single airfoil with 5557 points with space-filling curves a speedup of ~10X was reported with Quadro 2000 GPU.

B. Results of 2D Euler flow solver

Both sequential and parallel 2-D flow solvers were used to simulate transonic flow past airfoil at Mach number 0.8 and angle of attack 2°. Simulations are carried out on two sets cloud of points with sizes 5920 and 12240. The coarse grid is shown in fig. 5(a) and the Mach number contours on coarse grid are given in fig. 5(b). It can be observed from the figure that the shock wave on upper surface is captured well by the solver. The 2D flow solver was executed for 20,000 iterations

Table I. Comparison of aerodynamic coefficients of 2D Euler Flow Solver

No of Nodes	CL		CD	
	Seq. CPU	*Par. GPU*	*Seq. CPU*	*Par. GPU*
5920	0.5094279 23720567	0.50942792 3720567	0.03401515 7582313	0.03401515 7582312
12240	0.4995640 6 9417092	0.49956406 9417002	0.03514199 7567529	0.03514199 7567558

Table II. Performance of parallel code of 2D Euler Flow Solver

S.No	No of Nodes	T_s	T_p	CUDA Config	T_s/T_p
1.	5920	599.132	48.449	grid size: 24 block size: 256	12.366
2.	12240	1235.680	104.840	grid size: 48 block size: 256	11.790

Table III. Performance of parallel code of 3D Euler Flow Solver

S.No.	No of Nodes	Ts	Tp	CUDA config	Ts/Tp
1.	42664	8968.205	552.888	grid size: 167 block size: 256	16.22

and the aerodynamic Lift (CL) and drag (CD) coefficients are compared in Table I.

The aerodynamic coefficients are plotted with iteration in fig. 5(c). The results attain steady state at 10,000 iterations and the results from both solvers compare exactly with each other. The residue history is given in fig. 5(d). The residue has fallen 4 decades and they also compare well. Both aerodynamic coefficients compare till 14 digits between sequential and parallel solvers on both grids. This demonstrates the parallel solver works correctly and reproduces the solution that of sequential solver.

The performance of CUDA parallel version of software is compared with that of sequential version running on the CPU and the results are given in Table II. It can be noted that a performance improvement of 12X is obtained on an average.

C. Results of 3D Euler Flow Solver

Simulations are carried out on cloud of points with size 42664. The 3D flow solver was executed for 20,000 iterations and the aerodynamic forces are plotted in fig. 6(a) and moments are plotted in fig. 6(b) and fig. 6(c). The results attain steady state at 1200 iterations and the results from both solvers compare exactly with each other. The residue history is given in fig. 6(d). The residue has fallen 2 decades and they also compare well. Both aerodynamic coefficients compare till 14 digits between sequential and parallel solvers on both grids. The parallel version on GPGPU has shown a performance improvement of 16.22 times over the sequential counterpart on CPU, as shown in Table III.

VII. CONCLUSION

Grid free Euler solvers were converted into CUDA and MPI-GPU environments.

The results obtained with CUDA parallel version of the software over GPGPU were found to be identical to those of the sequential CPU version with good accuracy. The CUDA application has demonstrated a peak performance improvement of 16X. Though load balancing could not be achieved at the application level, the load balancing performed by the CUDA frame work of GPGPU produced good results. CFD applications based on structured grid may offer better performance improvement, as they facilitate coalesced memory access. The master-slave architecture can be effectively used for other accelerators also such as Intel© Xeon Phi® for good runtime performance of applications.

REFERENCES

1. Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, “Pattern-Oriented Software Architecture - Volume 1”, John Weley & Sons Publication 1996.
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Elements of Reusable Object-Oriented Software”, Adison Wesley Publication, 1994
3. Amdahl G.M., “Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities”, Proceeding of American Federation of Information Processing Societies Conference, AFIPS Press, pp. 483-485, 1967
4. Brockschmidt K, “Inside OLE 2”, Microsoft Press Publication, 1994
5. “The Common Object Request Broker: Architecture and Specification”, Object Management Group, OMG Document Number 91.12.1, Revision 1.1, 1992
6. Michael Wooldridge, “Agent-Based Software Engineering”, Mitsubishi Electric Digital Library Group, UK, 1997.
7. Murthy V. K. “Knowledge-Based Intelligent Information and Engineering Systems”, Lecture Notes on Computer Science, vol. 3681, Springer Publisher.
8. Javier Gonzalez-Sanchez, Maria Elena Chavez-Echeagaray, Robert Atkinson, Winslow Burleson, “ABE: An Agent-Based Software Architecture for A Multimodal Emotion Recognition Framework”, Proceedings of Ninth working IEEE/IFIP Conference on Software Architecture, 2011.
9. Nicholas R Jennings, “On Agent-Based Software Engineering”, Elsevier Publications, Journal Artificial Intelligence, vol.117, pp. 277-296, 2000
10. Ghassan Beydoun, Graham Low, Paul Bogg, “Suitability assessment framework of agent-based software architectures”, Faculty of Engineering and Information Sciences Papers, Information and Software Technology, pp. 673-689, 2013
11. Gulnara Zhabelova, “Software Architecture and Design Methodology for Distributed Agent-based Automation of Smart Grid”, Electrical and Electronic Engineering Department, University of Auckland, 2013
12. Etzioni O, “Moving up the information food chain: Deploying softbots and the world-wide web”, Proceedings of Thirteenth National conference on Artificial Intelligence (AAAI-96), 1996
13. Suresh S., Omkar S.N., Mani V., “Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations”, IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 1, 2005
14. Sahni S., Vairaktarakis G., “The master-slave paradigm in parallel computer and industrial settings”, Journal of Global Optimization Vol. 9, pp. 357-377, 1996.
15. Phil Brooks, “Master-Slave Pattern for Parallel Compute Services”, Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS), 1996
16. Srinivas J. V. S., Bhogendra Rao P. V. R. R., Prof. V. Kamakshi Prasad, “Parallel Implementation of Backpropagation on Master Slave Architecture”, Proceedings of IEEE International Conference on Computational Intelligence and Multimedia Applications, (ICCIMA 2007), 2007
17. Bhogendra Rao P. V. R. R., Shashank S. S., “Parallelization of Synthetic Aperture Radar (SAR) Image Formation Algorithm”, Proceedings of First International Conference on Computational Intelligence and Informatics, (ICCII 2016), Advances in Intelligent Systems and Computing, vol. 507, Springer Publications, 2016
18. Anandhanarayanan K: Development of 3D grid-free solver and its applications to multi-body aerospace vehicles, Defence Science Journal, Vol. 60, No. 6, pp. 653-662, 2010
19. NVIDIA CUDA Reference Manual, 3rd ed., NVIDIA Inc., Aug. 2010.
20. CUDA C Programming Guide, 6th ed., NVIDIA Inc., Aug. 2014.
21. Ma Z. H., Want. H., Pu S. H.: GPU computing of compressible flow problems by a meshless method with space-filling curves, Journal of Computational Physics, vol. 263, pp. 113--135, 2016

AUTHOR’S PROFILE

Dr PVRR Bhogendra Rao has been working as scientist in DRDL, DRDO for 27 years. He received his M.Tech and Ph.D from JNTU, Hyderabad. He worked in design, development, testing and deployment of real-time C4I systems. His areas of interest include pattern oriented design, real-time systems, fault tolerant systems, parallel and distributed computing