EXPLORING THE CONTRAST ON GPGPU COMPUTING THROUGH CUDA AND OPENCL

By

BALA DHANDAYUTHAPANI VEERASAMY * G. M. NASIRA **

* Research Scholar, Manonmaniam Sundaranar University, Tirunelveli, Tamilnadu, India.
** Assistant Professor, Computer Science, Chikkanna Govt Arts College Tirupur, Tamilnadu, India.

ABSTRACT

In the recent years, Multi-core to Many-core processors computing became most significant in High Performance Computing (HPC). Increasing parallelism rather than increasing clock rate has become the primary engine of processor performance growth and this trend is likely to continue. Particularly, today's Graphics Processing Units (GPU) became most significant in favour of HPC. General Purpose GPU (GPGPU) computing has allowed the GPU to emerge as successful co-processors that can be employed to improve the performance of many different non-graphical applications with high parallel requirements make them suitable for many HPC workloads. CUDA and OpenCL offer two different interfaces for programming GPUs. OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from different vendors, while CUDA is specific to NVIDIA GPUs. In this research paper, the authors explored the contrast between CUDA and OpenCL, helps the HPC programmers to familiarize with GPGPU.

Keywords: CPU, CUDA, Java, jCuda, jocl, GPGPU, GPU, OpenCL.

INTRODUCTION

Today's computing environments [1] are becoming more multifaceted, exploiting the capabilities of a range of multi-core microprocessors, central processing units (CPUs), digital signal processors, reconfigurable hardware (FPGAs), and graphic processing units (GPUs). Presented with so much heterogeneity, the process of developing efficient software for such a wide array of architectures poses a number of challenges to the programming community.

The Central Processing Unit (CPU) of a computer is a piece of hardware that carries out the instructions of a computer program. It performs the basic arithmetical, logical, and input/output operations of a computer system. A CPU has multiple execution cores on processing units called multi-core. The multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. A different category of computing problem [2], parallel computing has until recently remained the area of large server clusters and exotic supercomputers. Standard CPU architecture excels at managing many discrete tasks, but is not particularly efficient at processing tasks that can be divided into many smaller elements and analyzed in parallel. This is exactly the type of problem solved by Graphics Processing Units (GPU). The GPU has great potential for solving such problems quickly and inexpensively. GPU computing [7-9] makes supercomputing possible with any PC or workstation and expands the power of server clusters to solve problems that were previously not possible with existing CPU clusters. The goal of computing with GPUs [7-9] is to apply the tremendous computational power inherent in the GPU to solve some of the most difficult and important problems in high performance computing.

The programmable units [3] of the GPU result a Single Program Multiple Data (SPMD) programming model. For efficiency, the GPU processes many elements (vertices or fragments) in parallel using the same program. Each element is independent from the other elements, and in the base programming model, elements cannot communicate with each other. All GPU programs must be structured in this way: many parallel elements each processed in parallel by a single program. Each element can operate on 32-bit integer or floating point data with a reasonably complete general-purpose instruction set.

Elements can read data from a shared global memory and with the newest GPUs, also write back to arbitrary locations in shared global memory. This programming model is well suited to straight-line programs, as many elements can be processed in lockstep running the exact same code. Code written in this manner is Single Instruction, Multiple Data (SIMD). As shader programs have become more complex, programmers prefer to allow different elements to take different paths through the same program, leading to the more general SPMD model [10]. In a multiprocessor system, task parallelism is achieved when each processor executes different thread on the same or different data. In a multiprocessor system, each processor performs the same task on different pieces of distributed data. The following Table 1 shows the difference between CPU and GPU.

General-Purpose computation on Graphics Processing Units (GPGPU) [4] is also known as GPU Computing. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations.

GPU [5] have become important in providing processing power for high performance computing applications. CUDA and Open Computing Language (OpenCL) are

CPU	GPU
Serial workloads Task parallel workloads	Data parallel workloads
Vendors : Intel, AMD	Vendors : NVidia, AMD, ATI
4-8 cores (e.g. Inter Core i7)	1-512 cores (e.g. NVidia Fermi)
Low latency	Very high latency
Really fast caches (great for data reuse)	Lots of math units
Fine branching granularity	Fast access to onboard memory
Lots of different processes/threads	Run a program on each fragment/vertex
High performance	High throughput
MIMD architecture	SIMD architecture

Table 1. Comparison between CPU and GPU

two interfaces for GPGPU computing, both presenting similar features but through different programming interfaces. CUDA is a proprietary API and set of language extensions that works only on NVIDIA's GPUs. OpenCL, by the Khronos Group, is an open standard for parallel programming using Central Processing Units (CPUs), GPUs, Digital Signal Processors (DSPs), and other types of processors.

CUDA can be used in two different ways,

- via the runtime API, which provides a C like set of routines and extensions, and,
- via the driver API, which provides lower level control over the hardware but requires more code and programming effort. Both OpenCL and CUDA call a piece of code that runs on the GPU a kernel.

Setting up the GPU [6] for kernel execution differs substantially between CUDA and OpenCL. Their APIs for context creation and data copying are different, and different conventions are followed for mapping the kernel onto the GPU's processing elements. These differences could affect the length of time needed to code and debug a GPU application, but here we mainly focus on runtime performance differences. OpenCL promises a portable language for GPU programming, capable of targeting very dissimilar parallel processing devices. Unlike a CUDA kernel, an OpenCL kernel can be compiled at runtime, which would add to an OpenCL's running time. On the other hand, this just-in-time compile may allow the compiler to generate code that makes better use of the target GPU. CUDA, on the other hand, is developed by the same company that develops the hardware on which it executes, so one may expect it to better match the computing characteristics of the GPU, offering more access to features and better performance. Considering these factors, it is of interest to compare OpenCL's performance to that of CUDA in a real-world application.

1. CUDA - Compute Unified Device Architecture

GPU hardware [11] is radically different than CPU hardware. Notice the GPU hardware consists of a number of key blocks such as memory (global, constant, shared), Streaming Multiprocessors (SMs) and streaming

processors (SPs). The main thing to notice here is that a GPU is really an array of SMs, each of which has N cores (8 in G80 and GT200, 32–48 in Fermi, 8 plus in Kepler). This is the key aspect that allows scaling of the processor. A GPU device consists of one or more SMs. Add more SMs to the device and you make the GPU able to process more tasks at the same time, or the same task quicker, if you have enough parallelism in the task.

To a CUDA programmer, the computing system consists of a host, which is a traditional central processing unit (CPU), such as an Intel architecture microprocessor in personal computers today, and one or more devices, which are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Because data parallelism plays such an important role in CUDA, we will first discuss the concept of data parallelism before introducing the basic features of CUDA.

A CUDA [12-13] program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. A CUDA program is a unified source code encompassing both host and device code. The NVIDIA_ C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures. The device code is typically further compiled by the nvcc and executed on a GPU device. In situations where no device is available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU using the emulation features in CUDA Software Development Kit (SDK). We have showed some of familiar NVidia GPUs are shown in Table 2.

The steps in executing CUDA Programs are listed here under:

1. Device Initialization
2. Device memory allocation
3. Copies data to device memory
4. Executes kernel (Calling __global__ function)
5. Copies data from device memory (retrieve results)

2. OpenCL – Open Computing Language

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. OpenCL [14-15] is a heterogeneous programming framework that is managed by the nonprofits technology consortium Khronos Group. OpenCL is a framework for developing applications that execute across a range of device types made by different vendors. It supports a wide range of levels of parallelism and efficiently maps to homogeneous or heterogeneous, single- or multiple-device systems consisting of CPUs, GPUs, and other types of devices limited only by the imagination of vendors. The

GPU	G80	GT200	Fermi
Transistors	681 millions	1.4 billion	3.0 billion
CUDA Core	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops/clock	256 FMA ops/clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Special Function Units (SFUs)/SM	2	2	4
Warp Schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (Per SM)	None	None	Configurable 48 KB or 16 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Table 2. Example NVidia GPUs

OpenCL definition offers both a device-side language and a host management layer for the devices in a system.

The device-side language is designed to efficiently map to a wide range of memory systems. The host language aims to support efficient plumbing of complicated concurrent programs with low overhead. Together, these provide the developer with a path to efficiently move from algorithm design to implementation. OpenCL provides [14-15] parallel computing using task-based and data-based parallelism. It currently supports CPUs that include x86, ARM, and PowerPC, and it has been adopted into graphics card drivers by both AMD (called the Accelerated Parallel Processing SDK) and NVIDIA. Support for OpenCL [14-15] is rapidly expanding as a wide range of platform vendors have adopted OpenCL and support or plan to support it for their hardware platforms. These vendors fall within a wide range of market segments, from the embedded vendors (ARMand Imagination Technologies) to the HPC vendors (AMD, Intel, NVIDIA, and IBM). The architectures supported include multi-core CPUs, throughput and vector processors such as GPUs, and fine grained parallel devices such as FPGAs. Most important, OpenCL's cross-platform, industrywide support makes it an excellent programming model for developers to learn and use, with the confidence that it will continue to be widely available for years to come with ever-increasing scope and applicability.

The OpenCL C programming language [14-15] is used to create programs that describe data-parallel kernels and tasks that can be executed on one or more heterogeneous devices such as CPUs, GPUs, and other processors referred to as accelerators such as DSPs and the Cell Broadband Engine (B.E.) processor. An OpenCL program is similar to a dynamic library, and an OpenCL kernel is similar to an exported function from the dynamic library. Applications directly call the functions exported by a dynamic library from their code. Applications, however, cannot call an OpenCL kernel directly but instead queue the execution of the kernel to a command-queue created for a device. The kernel is executed asynchronously with the application code running on the host CPU. The OpenCL specification [14] is defined in four parts, called models that can be summarized as follows:

2.1 Platform model

This model specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices). It defines an abstract hardware model that is used by programmers when writing OpenCL C functions (called kernels) that execute on the devices.

2.2 Execution model

This model defines how the OpenCL environment is configured on the host and how kernels are executed on the device. This includes setting up an OpenCL context on the host, providing mechanisms for host–device interaction, and defining a concurrency model used for kernel execution on devices. OpenCL application runs on a host which submits work to the compute devices

Context

The environment within which work-items executes, includes devices and their memories and command queues

Program

Collection of kernels and other functions (Analogous to a dynamic library)

Kernel

the code for a work item. Basically a C function

Work item

the basic unit of work on an OpenCL device

2.3 Memory model

This model defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies, although this has not limited adoptability by other accelerators.

- Private Memory–Per work-item
- Local Memory–Shared within a workgroup
- Global/Constant Memory–Visible to all workgroups
- Host Memory–On the CPU

2.4 Programming model

This model defines how the concurrency model is mapped to physical hardware. OpenCL uses Dynamic/Runtime compilation model:

- The code is complied to an Intermediate Representation (IR)
- Usually an assembler or a virtual machine
- Known as offline compilation
- The IR is compiled to a machine code for execution.
- This step is much shorter.
- It is known as online compilation

The steps in executing OpenCL Programs are listed here under:

1. Query host for OpenCL devices

2. Create a context to associate OpenCL devices

3. Create programs for execution on one or more associated devices

4. Select kernels to execute from the programs

5. Create memory objects accessible from the host and/or the device

6. Copy memory data to the device as needed

7. Provide kernels to command queue for execution

8. Copy results from the device to the host

3. Contrasts between CUDA and OpenCL

The OpenCL API is a C [14-15] with a wrapper API that is defined in terms of the C API. There are third-party bindings for many languages, including Java, Python and .NET. The code that executes on an OpenCL device, which in general is not the same device as the host CPU, is written in the OpenCL C language. OpenCL C is a restricted version of the C99 language with extensions appropriate for executing data-parallel code on a variety of heterogeneous.

CUDA encourages the use of scalar code in kernels. While this works in OpenCL as well, depending on the desired target architecture, it may be more efficient to write programs operating on OpenCL's vector types, such as float, as opposed to pure scalar types. This is useful for both AMD CPUs and AMD GPUs, which can operate efficiently on vector types. OpenCL also provides flexible swizzle/broadcast primitives for efficient creation and rearrangement of vector types. CUDA does not provide rich facilities for task parallelism, and so it may be beneficial to think about how to take advantage of OpenCL's task parallelism as you port your application.

CUDA [14] is a parallel computing framework designed only for NVIDIA's GPUs, and OpenCL is a standard designed for diverse platforms including CUDA-enabled GPUs, some ATI-GPUs, multi-core CPUs from Intel and AMD, and other processors such as the Cell Broadband Engine. OpenCL shares a range of core ideas with CUDA: they have similar platform models, memory models, execution models, and programming models. To a CUDA/OpenCL programmer, the computing system consists of a host (typically a traditional CPU), and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units. There also exists a mapping between CUDA and OpenCL in memory and execution terms, are presented in Table 3. Additionally, their syntax for various keywords and built-in functions are fairly similar to each other. Therefore, it is relatively straightforward to translate CUDA programs to OpenCL programs.

4. Our researches towards parallel computing

Parallel computers can be separated into two kinds of major categories, they are control flow and data flow. The control flow parallel computers are called task parallelism. Programs decomposed into individual parts statements, methods can be run in parallel. Some of our earlier researches concentrated on task parallelism; all these task parallelism can be executed on MPMD [16]. they are Setting CPU Affinity in Windows Based SMP Systems Using Java [17], Parallel: One Time Pad using Java [18], JNT - Java Native Thread for Win32 Platform [19], Java Native Pthread for Win32 Platform [20], Overall Aspects of Java Native Threads on Win32 Platform [21], Performance Analysis of Java NativeThread and NativePthread on Win32 Platform [22], Native Pthread on Android Platform using Android NDK [23]. The data parallelism uses the input data to some operation as the means to partition into smaller pieces either large amount of data to process or combination of both. Data is divided up among the

C for CUDA Terms	OpenCL Terms
Thread	Work-item
Thread block	Work-group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory
Qualifiers	
__global__function (callable from host, not callable from device)	__kernel function (callable from device, including CPU device)
__constant__ variable declaration	__constant variable declaration
__device__variable declaration	__global variable declaration
__shared__variable declaration	__local variable declaration
Indexing	
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id
Synchronization	
__syncthreads()	barrier()
__threadfence_block()	mem_fence()
Import API Objects	
CUdevice	cl_device_id
CUcontext	cl_context
CUmodule	cl_program
CUfunction	cl_kernel
CUdeviceptr	cl_mem
Important API Calls	
cuDeviceGet()	clGetContextInfo()
cuCtxCreate()	clCreateContextFromType()
cuModuleGetFunction()	clCreateKernel()
cuMemAlloc()	clCreateBuffer()
cuMemcpyHtoD()	clEnqueueWriteBuffer()
cuMemcpyDtoH()	clEnqueueReadBuffer()
cuParamSeti()	clSetKernelArg()
cuLaunchGrid()	clEnqueueNDRangeKernel()
cuMemFree()	clReleaseMemObj ()

Table 3. Common Terms used in C for CUDA and Open CL

available hardware processors in order to achieve parallelism; this can be achieved through SPMD [10]. One of our earlier researches concentrated on data parallelism on CPU that is Java can facilitate Intel TBB through JNI [24]. It can exploit painless usage of Intel TBB parallel algorithms that can be used in Java.

Java programmer needs to take advantage of a GPUs massive parallelism potential unless she fiddles with Java Native interface. We can propose GPGPU computing for Java to facilities either for CUDA or OpenCL through JNI. Of course there are some Java tools out there that ease the pain of GPGPU Java programming [25-26]. The two most popular are jocl [27] and jcuda [28]. With these tools we still have to write C/C++ code but at least that would be only for the code that will be executed in the GPU, minimizing the effort considerably.

The jocl [27] contains Java bindings for OpenCL, the Open Computing Language. OpenCL allows writing programs for heterogeneous platforms that utilize CPUs or GPUs. The following implementations of OpenCL are currently available:

- The AMD APP SDK with support for OpenCL 1.2, version 2.7
- The NVIDIA drivers with support for OpenCL 1.1
- OpenCL in Apple Snow Leopard
- Intel OpenCL SDK

This library offers Java-Bindings for OpenCL that are very similar to the original OpenCL API. The functions are provided as static methods, and semantics and signatures of these methods have been kept consistent with the original library functions, except for the language-specific limitations of Java. The OpenCL API may be very verbose at some points, and this is not hidden or simplified, but simply offered by JOCL as-it-is.

As an alternative for Java developers who want to benefit from the computing power of their GPU without having to learn OpenCL, AMD has published Aparapi: It allows a seamless integration of GPU workloads into Java Code. The Java bytecode will be converted into OpenCL code and executed on the GPU, transparently for the user. Even if there is no OpenCL implementation, the same code will still run on the CPU in a Java Thread Pool, taking advantage of multiple CPU cores.

The jcuda [28] contains Java bindings for CUDA for NVIDIA® CUDA™ and related libraries. To use these libraries, you need a CUDA-enabled GPU device and the NVIDIA driver with CUDA support and the CUDA Toolkit from the NVIDIA website. The APIs of the libraries on this site have been kept close to the original APIs. The functions of all libraries are provided as static methods, and semantics and signatures of these methods have been kept

consistent with the original library functions, except for the language-specific limitations of Java.

Conclusion

The heterogeneous computer system with multi-core computing (on CPU) to many-core computing (on GPU) became most significant in HPC. Massively parallel computation can be supported on recent GPU in favour of HPC. The recent GPUs trends are moving towards to multi GPUs [29]. CUDA and OpenCL offer two different interfaces for General Purpose programming GPUs. OpenCL is an open standard that can be used to program central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), Field-Programmable Gate Arrays (FPGAs) and other processors from different vendors adopted by Apple, Intel, Qualcomm, Advanced Micro Devices (AMD), Nvidia, Xilinx, Altera, Samsung, Vivante, Imagination Technologies and ARM Holdings, while CUDA is specific to NVIDIA GPUs. Java Native Interface assist java programmer to create native API for Java. Through JNI we can avail OpenCL and CUDA programming for Java. However it has been identified that jocl and jcuda are the Java binding for GPGPU computing though this binding are not up-to-date to the present supported library on OpenCL or CUDA. This research paper, we explored the contrast between CUDA and OpenCL, helps the HPC programmers to familiarize with GPGPU.

References

[1]. Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa, (2012). *Heterogeneous Computing with OpenCL*, 2012, Elsevier Inc.

[2]. NVIDIA Tesla (2007). GPU Computing Technical Brief, GPU Compute Tech Brief, 2007.

[3]. John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, (2008). GPU Computing, *IEEE Xplore*.

[4]. GPU Computing, http://gpgpu.org/about, accessed on 18 march 2015.

[5]. Kamran Karimi Neil G. Dickson Firas Hamze, (2011). A Performance Comparison of CUDA and OpenCL.

[6]. Jianbin Fang, Varbanescu, A.L. ; Sips, H.,(2011). A Comprehensive Performance Comparison of CUDA and OpenCL, *IEEE Xplore*.

[7]. John Nickolls, William J. Dally, (2010). GPU Computing Era, *IEEE*.

[8]. Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar, (2012). GPGPU Processing In CUDA Architecture*, Advanced Computing: An International Journal (ACIJ)*, Vol.3, No.1.

[9]. John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, (2008). GPU Computing, *IEEE*.

[10]. Bala Dhandayuthapani Veerasamy, (2010). Concurrent Approach to Flynn's SPMD Classification, *International Journal of Computer Science and Information Security*, Vol. 7, No. 2.

[11]. Shane Cook, (2013). CUDA Programming A Developer's Guide to Parallel Computing with GPUs, Elsevier Inc.

[12]. David B. Kirk and Wen-mei W. Hwu, (2010). Programming Massively Parallel Processors A Hands-on Approach, Elsevier.

[13]. Michael Garland, Joshua Anderson, Jim Hardwich, Scott Mroton, Everett Phillops, Vasily Vokkov, (2008). Parallel Computing Experiences with CUDA, IEEE.

[14]. Lee Howes and Aaftab Munshi, (2014). The OpenCL Specification, Khronos OpenCL

[15]. AMD Accelerated Parallel Processing OpenCl Optimization Guide, 2014, Advanced Micro Devices, Inc.

[16]. Bala Dhandayuthapani Veerasamy, (2010). Concurrent Approach to Flynn's MPMD Classification through Java, *International Journal of Computer Science & Network Security*, Vol.10, No.2.

[17]. Bala Dhandayuthapani Veerasamy, Dr. G.M. Nasira, (2012). Setting CPU Affinity in Windows Based SMP Systems Using Java, *International Journal of Scientific & Engineering Research*, Vol. 3, No. 4, pp 893-900, Texas, USA, ISSN 2229-5518.

[18]. Bala Dhandayuthapani Veerasamy, Dr. G.M. Nasira, (2012). Parallel: One Time Pad using Java, International Journal of Scientific & Engineering Research,

Vol. 3, No. 11, pp 1109-1117, Texas, USA, ISSN 2229-5518.

[19]. Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, (2013). JNT - Java Native Thread for Win32 Platform, *International Journal of Computer Applications*, Vol. 70, No. 24, pp 1-9, Foundation of Computer Science, New York, USA, ISSN 0975 - 8887.

[20]. Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, (2014). Java Native Pthreads for Win32 Platform, *World Congress on Computing and Communication Technologies (WCCCT'14)*, pp.195-199, Feb. 27 2014-March 1 2014, Tiruchirappalli, published in IEEE Xplore, ISBN: 978-1-4799-2876-7.

[21]. Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, (2014). Overall Aspects of Java Native Threads on Win32 Platform, *Second International Conference on Emerging Research in Computing, Information, Communication and Applications (ERCICA-2014)*, Vol., No., pp. August 01-02, 2014, Bangalore, published in ELSEVIER in India.

[22]. Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, Performance Analysis of Java NativeThread and NativePthread on Win32 Platform, *International Journal of Computational Intelligence and Informatics*, ISSN: 2349-6363 (on Process).

[23]. Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, Native Pthread on Android Platform using Android NDK, *Karpagam Journal of Computer Science (KJCS)*, ISSN : 0973-2926 (On Process).

[24]. Bala Dhandayuthapani Veerasamy and Dr. G.M. Nasira, Java Native Intel Thread Building Blocks for Win32 Platform, *Asian Journal of Information Technology*, Vol. 13, No. 8, pp 431-437, Medwell Publishing, ISSN 1682-3915 (Print).

[25]. Jorge Docampo, Sabela Ramos, Guillermo L. Taboada, Roberto R. Exp´osito, Juan Touri˜no, Ram´on Doallo, (2013). Evaluation of Java for General Purpose GPU Computing, IEEE

[26]. Yonghong Yan, Max Grossman, and Vivek Sarkar, (2009). *JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA, LNCS 5704*, pp. 887–899, 2009, Springer-Verlag Berlin Heidelberg.

[27]. Java bindings for OpenCL, http://www.jocl.org, accessed on 18 March 2015.

[28]. Java bindings for CUDA, http://www.jcuda.org, accessed on 18 March 2015.

[29]. Long Chen, Oreste Villa,Guang R. Gao, (2011). Exploring Fine-Grained Task-based Execution on Multi-GPU Systems, IEEE.

ABOUT THE AUTHORS

Bala Dhandayuthapani Veerasamy is currently working as an IT Lecturer in Shinas College of Technology, Oman. He received his B.Sc in Computer Science from Bharathidasan University in 2000. He received his first master degree M.S in Information Technology from Bharathidasan University in 2002 and he received his second master degree M.Tech in Information Technology from Allahabad Agricultural Institute of Deemed University in 2005. Presently, he is pursuing part-time external PhD in the areasof Information Technology from Manonmaniam Sundaranar University. So far, he involved more than twenty peer reviewed research papers on various international conferences and Journals.

G M Nasira is working as an Assistant Professor in the Department of Computer Science, at Chikkannna Government Arts College, Tiruppur. She got her B.Sc (Computer Science) from Madras University, MCA from Bharathidasan University, B.Ed and M.Phil from Bharathiyar University. She got her Ph.D. in Computer Science from Mother Teresa Women's University, Kodaikanal with the specilisation of Artificial Neural Networks. She has published so far 12 research papers in referred Journals and presented 40 papers in various conferences and seminars in addition she has also authored a book titled Fundamentals of Middleware Technologies and Web Technologies.