ISSN: 2278 - 1323
International Journal of Advanced Research in Computer Engineering and Technology (IJARCET)
Volume 2, Issue 6, June 2013

High-Performance Computing using GPUs

Rakesh Kumar K. N, Hemalatha V, Shivakumar K. M, Basappa B. Kodada

Abstract— **In the last few years, emergence of High-Performance Computing has largely influenced computer technology in the field of financial analytics, data mining, image/signal processing, simulations and modeling etc. Multi-threading, hyper-threading and other parallel programming technologies, multicore machines, clusters etc. have helped achieve high performance with high availability and high throughput. However, hybrid clusters have been gaining increased popularity these days due to the performance efficient, accelerated computation service they provide with the help of accelerators like GPUs (Graphics Processing Units), FPGAs (Field Programmable Gate Arrays), DSPs (Digital Signal Processors) etc. Amongst the accelerators, GPUs are most widely used for general purpose computations which require high speed and high performance as GPUs are highly parallel, multithreaded, manycore processors with tremendous computational power. In this paper, the hardware architecture of GPUs is discussed with light on how it is favorable to be used for general purpose computations. The programming models which are generally used these days for GPU computing and comparison of those models are discussed. Also a comparison of GPUs and CPUs in terms of both architecture and performance with the help of a benchmark is made and finally concluded with results.**

Index Terms— **GPU computing, GPGPU, Comparison of CPU and GPU computation performance, High performance computing**

I. Introduction

Hybrid clusters have been gaining increased attention these days with the high performance services they offer. Different types of accelerators, attached to the compute nodes like GPUs (Graphics Processing Units), FPGAs (Field Programmable Gate Arrays), DSPs (Digital Signal Processors) enhance the efficiency by allowing parallel executions and provide high throughput.

Many-core graphics processors and FPGAs gain an increasingly important position concerning the advancements on modern computing systems. Clusters augmented with application accelerators have been evolving as competitive high performance computing systems these days. Adding Field-Programmable Gate Array (FPGA) coprocessors to the clusters can boost application performance, reduce power consumption and the total cost of ownership. The Graphical Processing Unit (GPU) with a very high arithmetic density and performance per price ratio is a good platform for the scientific application acceleration. Also, GPUs are highly parallel, multithreaded, manycore processors with tremendous computational power. Because of these reasons, GPUs and FPGAs are mostly recommended as accelerators to build hybrid cluster.

Apart from being used as a coprocessor in cluster environment, GPUs are used with a single system also. GPUs when employed on general computations rather than graphics specific computations are termed as GPGPUs (General Purpose Graphics Processing Units). Scientific research related computations, image/sound/signal processing, real-time simulations etc. which require diverse, huge and highly parallel processing are these days performed using GPGPUs.

A traditional CPU includes no more than a handful of cores. But a GPU has a massively parallel array of integer and floating-point processors, as well as dedicated, high-speed memory. This drags the attention towards general purpose computation on a GPU. But performing computations on a GPU is beneficial only at the cost of memory access time. Data must be sent from the CPU to the GPU before calculation and then retrieved from it afterwards. Because a GPU is attached to the host CPU via the PCI Express bus, the memory access is slower than with a traditional CPU. This means that the overall computational speedup is limited by the amount of data transfer that occurs in the algorithm thus making memory access a bottleneck. But this can be compensated if the algorithm is restructured to reduce the data transfer rates.

In section II, a brief survey on the systems which use GPUs as accelerators is explained. In section III, the architecture of GPU is explained in general with the help of a diagram of Nvidia Fermi hardware architecture. A general discussion on how the basic hardware organization of memory and other functional units of the device help achieve parallelism inherently thereby entertaining the general purpose computations on GPUs is made. In section IV, GPU computing is explained. The programming models or the frameworks used to program GPUs for user requirements are mentioned and explained with a brief introduction to the processing flow and methodology employed. CUDA and OpenCL are compared in brief. The last section covers the details on experiments we performed to compare the performance of CPUs and GPUs. A 2-D graph is plotted to explain the difference in computation times.

II. Systems Built Using GPUs

A heterogeneous computer cluster called Axel was developed with each node attached with multiple types of accelerators including GPUs. More emphasis is given to the performance and the scalability of applications across the network [1]. AMAX's ClusterMax SuperG GPGPU clusters are powered by the NVIDIA Tesla 20-series GPU computing platforms, preinstalled with Redhat Enterprise Linux 5.x/6.x, 64-bit or SuSe Linux Enterprise Server 11x, CUDA 4.x Toolkit and SDK and a clustering software [2].

NCHPC Appro HyperPower GPU clusters allow scientific and technical professionals the opportunity to test and

experiment their ability to develop applications faster and to deploy them across multiple generations of processors [3]. Center for Manycore Programming, School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea developed SnuCore, a 16-node heterogeneous CPU/GPU cluster. SnuCL supports X86 CPUs, ARM CPUs, PowerPC CPUs, NVIDIA GPUs, AMD GPUs as compute devices [4]. Tianhe-1A epitomizes modern heterogeneous computing by coupling massively parallel GPUs with multi-core CPUs, enabling significant achievements in performance, size and power. The system uses 7,168 NVIDIA Tesla M2050 GPUs and 14,336 CPUs; it would require more than 50,000 CPUs and twice as much floor space to deliver the same performance using CPUs alone [5].

Other installations include GraphStream Inc., [6][7], a 160-node "DQ" GPU cluster at LANL [6][8] and a 16-node "QP" GPU cluster at NCSA [6][9], based on NVIDIA QuadroPlex technology. At NCSA two GPU clusters were deployed based on the NVIDIA Tesla S1070 Computing System: a 192-node production cluster "Lincoln" [10] and an experimental 32-node cluster "AC" [11], which is an upgrade from prior QP system. Both clusters went into production in 2009 [6].

III. GPU Architecture

GPUs were basically designed for graphics rendering and for a particular class of applications with the following characteristics where higher performance was achieved with the inherent hardware properties of the GPUs [12]. Later on, other applications with similar characteristics were identified and these days are being successfully mapped onto the GPUs.

A. *Large Computational Requirements*

Real-time rendering requires billions of pixels per second, and each pixel requires hundreds or more operations. GPUs must deliver an enormous amount of compute performance to satisfy the demand of complex real-time applications. This property makes GPU computing applicable for domains where computations are huge.

B. *Parallelism is substantial*

Applications which fall into computational domains where there is scope for achieving parallelism without any dependencies, are very well suited to be employed with GPUs. The graphics pipeline is well suited for parallelism. Operations on vertices and fragments are well matched to finegrained closely coupled programmable parallel compute units.

C. *High throughput*

High throughput at the cost of latency. GPU implementations of the graphics pipeline prioritize throughput over latency.

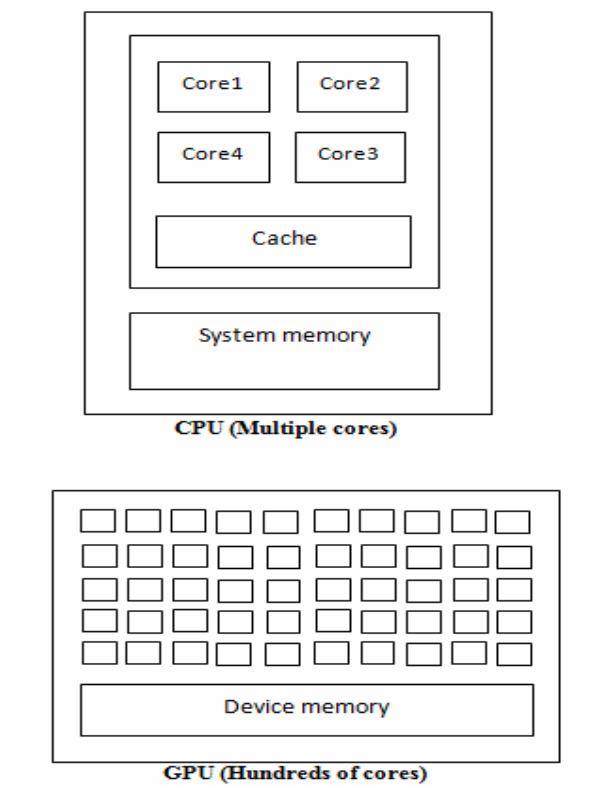

Figure I. Comparison of the number of cores on a CPU and a GPU.

Most of the applications related to scientific research including large dataset simulation modelling, signal/image/sound processing, real-time financial analytic systems etc. have the requirements matching the characteristics of GPUs. So, such computations are usually carried out with the help of GPUs as accelerators. Figure I shows the comparison of the number of cores on a CPU system and a GPU. A GPU has a massively parallel array of integer and floating-point processors, as well as dedicated, high-speed memory [13].

As already discussed, GPUs are built for different application demands. i.e., large, parallel computation requirements with emphasis on throughput rather than latency. With increased demands, the architecture of GPUs has been progressing more frequently to serve general purpose computation demands with improved hardware functionalities to enable parallelism with more number of functional units embedded along with better memory and improved and more advanced programming models to program the GPU applications with concern on reducing the delay or latency and increasing the throughput and parallelism.

Figure II is a simplified hardware block diagram for the NVIDIA Fermi GPU architecture. Fermi contains up to 512 general purpose arithmetic units known as Streaming Processors (SP) and 64 Special Function Units (SFU) for computing special transcendental and algebraic functions not provided by the SPs [14]. Memory load/store units (LDST), texture units (TEX), fast on-chip data caches, and a high-bandwidth main memory system provide the GPU with sufficient operand bandwidth to keep the arithmetic units

productive. Groups of 32 SPs, 16 LDSTs, 4 SFUs, and 4 TEXs compose a Streaming Multiprocessor (SM). One or more CUDA "thread blocks" execute concurrently on each SM, with each block containing 64–512 threads. The block diagram explains the GPU architecture in general. Any other GPU has a similar architecture with few variations. The latest Kepler for example, has some extra units with additional functionalities. The same holds good for AMD GPUs also. The only thing kept in mind while manufacturing a GPU for general purpose is that it serves as a massively parallel, fast, high throughput processing unit as an accelerator. Some interesting facts about the recent Kepler devices are SMX- delivers more processing performance and efficiency through innovative streaming multiprocessor design that allows a greater percentage of space to be applied to processing cores versus control logic, Dynamic Parallelism- simplifies GPU programming by allowing programmers to easily accelerate all parallel nested loops- resulting in a GPU dynamically spawning new threads on its own without going back to the CPU and Hyper-Q- slashes CPU idle time by allowing multiple CPU cores to simultaneously utilize a single Kepler GPU, dramatically advancing programmability and efficiency [15].

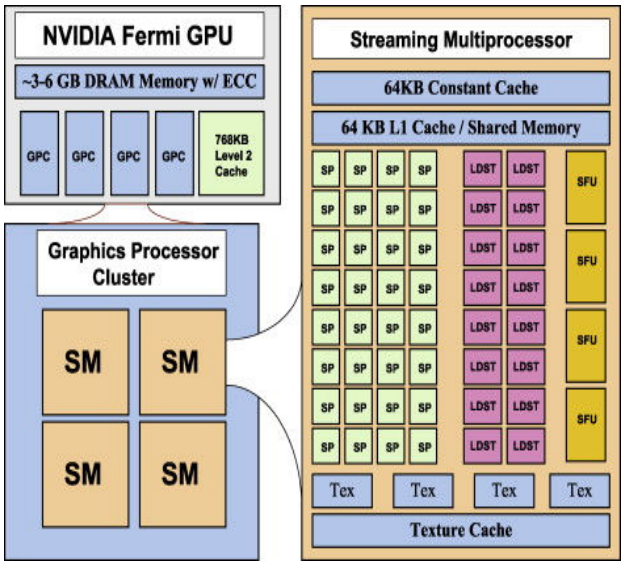

Figure II. NVIDIA Fermi GPU architecture (Diagram courtesy of NVIDIA).

IV. GPU COMPUTING

In the last section, the hardware architecture of GPU has been explained. The hardware architecture has evolved over years to match the requirement of the device to be used for general purpose computations along with graphics rendering. But hardware architecture itself cannot change the way GPUs are used for different applications. Hardware architecture just explains the functional units used and their interoperability. To achieve parallelism or higher performance, it is ultimately the programming methodology, important that utilizes the hardware resources available in the devices efficiently. In this section the programming model of GPU is explained. The programming model describes the methodology adopted at lower level to communicate with the hardware components.

A programming model allows users to program GPUs according to their requirements at a higher level, hiding the complexity of mapping the application level contexts to the subcomponents. Mapping user level code to working components mainly includes scheduling of tasks and distribution of data. GPUs support Single Instruction Multiple Data model with the help of "scatter-gather" approach. The GPU programs are structured according to the programming model so that the elements are run in parallel and data is distributed or scattered from shared global memory and the results are gathered onto the shared global memory.

The transfer of data from RAM to GPU memory and vice versa is the bottleneck outside of the GPU device and the care is taken to organize the operations with the help of the programming model in CPU-GPU program such that the kernel part of the algorithm is executed at once hence reducing the cost of communication and thereby reduce the communication-computation ratio.

The way kernels or the operations are to be executed on the device and data to be distributed among subcomponents of the device depends on the scheduling policy employed in the programming model scheduler module. The scheduler module decides how many and exactly which subcomponents are to be employed for the particular application. The scheduler in turn takes the decision dynamically on the number of subcomponents to be employed for a particular application based on the size of the execution units or execution matrix specified by the user in the GPU program. The GPU program usually contains two parts, CPU-part and GPU-part. CPU part is the normal CPU-execution part and GPU-part refers to the portion of program where there is a scope for parallelism and GPU is suitable to be worked on. This portion is called the kernel part and the kernel part has to be explicitly specified by the user with the help of certain keywords during programming with sizes for grids and blocks. With this, the user can specify the requirement on the number of execution units. But depending on the actual hardware architecture available, it is the duty of programming model to distribute the functional units with proper scheduling for elements of execution such that the resources demanded by the user in program are satisfied. An abstraction is provided by the programming model so that there does not exist any problem with the mapping between user program and underlying hardware architecture. For varying architectures, it is the responsibility of the programming systems to hide the inner details of the hardware and provide user the expected results.

Both AMD and NVIDIA have their own GPGPU programming systems. AMD announced and released their system in late 2006. CTM (Close to the metal) provides a low-level Hardware Abstraction Layer (HAL) for the R5XX and R6XX series of ATI GPUs. CTM HAL provides raw assembly-level access to the fragment engines (stream processors) along with an assembler and command buffers to control execution on the hardware. No graphics-specific features are exported through this interface [14]. Computation is performed by binding memory as inputs and outputs to the stream processors, loading an ELF binary, and defining a domain over the outputs on which to execute the

binary. AMD also offers the compute abstraction layer (CAL), which adds higher level constructs, similar to those in the Brook runtime system, and compilation support to GPU ISA for GLSL, HLSL, and pseudoassembly like Pixel Shader 3.0. For higher level programming, AMD supports compilation of Brook programs directly to R6XX hardware, providing a higher level programming abstraction than provided by CAL or HAL.

NVIDIA's CUDA is a higher level interface than AMD's HAL and CAL. Similar to Brook, CUDA provides a C-like syntax for executing on the GPU and compiles offline [14].

A. *Compute Unified Device Architecture*

Compute Unified Device Architecture (CUDA) is a programming model for NVIDIA GPUs. CUDA is a parallel computing platform that gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. C/C++ programmers use 'CUDA C/C++', compiled with 'nvcc', NVIDIA's LLVM-based C/C++ compiler, and Fortran programmers can use 'CUDA Fortran', compiled with the PGI CUDA Fortran compiler from The Portland Group.

Figure III explains the processing flow of CUDA [16]. GeForce 8800 is considered for the example. The processing flow is explained in four steps. As shown in the figure, in the first step the data is copied from main memory of the system to the GPU memory for processing. In the second step, instruction is sent from CPU to the GPU device to start processing. In third step, GPU executes the task with given data in parallel. In the last step, the result from GPU memory is copied to the main memory. CUDA APIs are used to transfer data to and from the device.

A Kernel is executed as a grid of thread blocks [18]. As shown in Figure IV, two kernels in an application can exist. The block size and grid size are specified by the user while the kernel is called. The threads share data memory space. A thread block is a batch of threads that can cooperate with each other by synchronizing their executions and efficiently sharing data through a low latency shared memory. However two threads from two different blocks cannot cooperate.

B. *OpenCL (Open Computing Language)*

OpenCL is a framework for writing programs that execute across heterogeneous platforms including CPUs, GPUs, DSPs etc. It includes a language based on C99 for writing kernels and APIs that are used to define and then control the platforms. OpenCL provides parallel computing using task-based and data-based parallelism.

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. It has been adopted by Intel, AMD, Nvidia, Altera, Samsung etc. Academic researchers have investigated automatically compiling OpenCL programs into application-specific processors running on FPGAs [19], and commercial FPGA vendors are developing tools to translate OpenCL to run on their FPGA devices [20] [21].

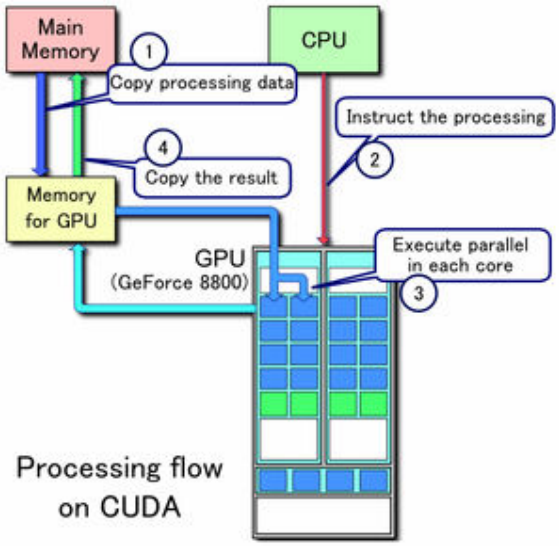

Figure III. Processing flow of CUDA (Diagram courtesy of NVIDIA)

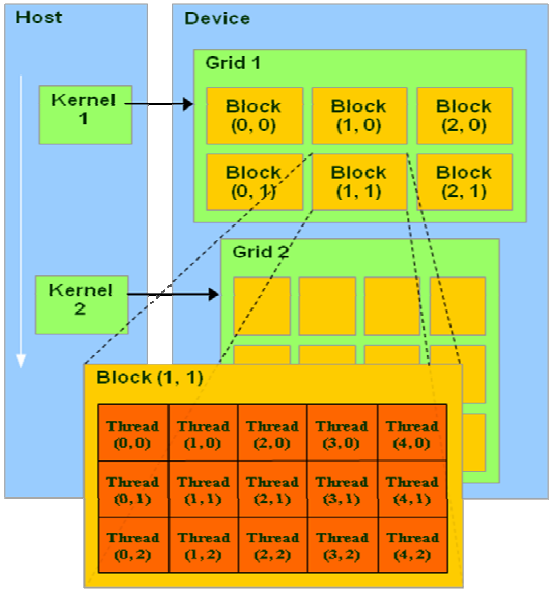

Figure IV. Kernels being executed as grids of thread blocks

Comparing CUDA and OpenCL, there is only one vendor of CUDA implementations i.e, Nvidia. But for OpenCL there are many including Nvidia, Apple, AMD, Intel etc [22]. Also, OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices, while CUDA is specific to NVIDIA GPUs. But as far as performance is concerned, CUDA is better in terms of transferring of data to and from GPU and kernel execution time [23].

V. Experiments To Compare The Performance Of CPU and GPU

To compare the performance of CPU and GPU we used Fast Fourier Transform (FFT) as a benchmark and applied FFT on an input digital image matrix and calculated the processing time on both CPU and GPU. The image size was increased in a loop and a graph was plotted for CPU and GPU

computations. As expected, the execution times on GPU for all sizes were less than the execution times on CPU for the same input sizes. The processor considered is Intel core i3 and GPU is NVIDIA GeForce 315M.

Figure V is the graph which depicts the scenario. X-axis represents the loop (each unit in x-axis represents a multiple of size of input matrix) and each unit in y-axis refers to a multiple of CPU-time (yellow line) and GPU-time (light green line).

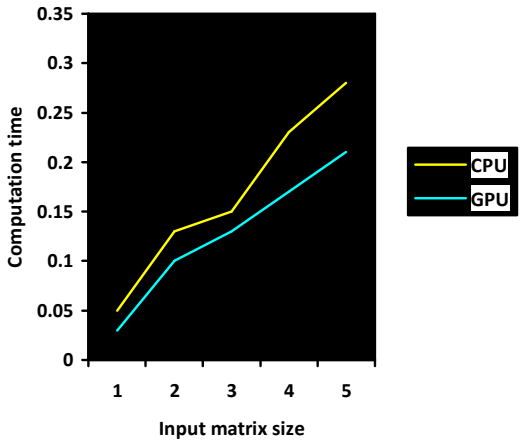

Figure V. Comparison of computation times for FFT on CPU and GPU

Here MATLAB is used for both CPU and GPU computation as using MATLAB for GPU computing helps accelerate applications with GPUs more easily than by using C or Fortran. With the familiar MATLAB language it is easy to take advantage of the CUDA GPU computing technology without having to learn the intricacies of GPU architectures or low-level GPU computing libraries [17]. With Parallel Computing Toolbox, it is possible to directly communicate with the GPUs and use the GPU-enabled MATLAB functions such as `fft`, `filter`, and several linear algebra operations. Also CUDA kernel integration in MATLAB is possible with Parallel Computing Toolbox. But to make it simple here, the MATLAB fft function is used directly. The size of the input matrix was further increased in loop to check the difference in computation times and it was noticed that the difference increased with the size. This proves that GPU computing is very useful in huge computation involving less data transfer and scope for parallel executions.

A mathworks team performed an experiment to solve second-order wave equations [13]. They ran a benchmark study and measured the amount of time the algorithm took to execute 50 time steps for grid sizes of 64, 128, 512, 1024, and 2048. The benchmark was run on an Intel Xeon Processor X5650 and NVIDIA Tesla C2050 GPU.

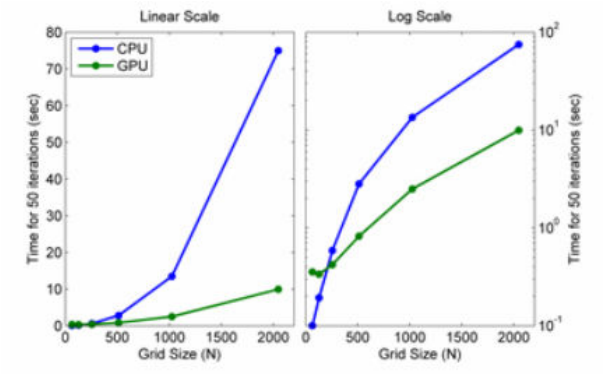

Figure VI. Benchmark results showing the time required to complete 50 time steps for different grid sizes, using linear scale (left) and a log scale (right).

For a grid size of 2048, the algorithm shows a 7.5x decrease in compute time as shown in Figure VI. The log scale plot shows that the CPU is actually faster for small grid sizes.

VI. Conclusion

A theoretical introduction to GPU computing, related work and applications are discussed briefly. The architecture of GPUs which is favourable to be used for general purpose computations, supporting parallelism is discussed. Organization of the device memory and other functional units are explained with the help of an Nvidia Fermi GPU architecture. The computing or programming models used to program GPUs for general purpose computations and processing flow of CUDA are discussed. The reason and idea behind developing such programming models and their importance in achieving abstraction and higher performance are explained in detail. A brief comparison of CUDA and OpenCL is made by considering both performance and portability as parameters. In the last section, experimental results for comparison of GPU and CPU performance by considering FFT as a benchmark is plotted and thus concluded that GPUs are more performance efficient than CPUs for huge computations involving good scope for achieving parallelism and less data transfers between system memory and the device. Also, an experiment performed by a Mathworks team is mentioned which shows that CPU performs better than a GPU for very small grid sizes. However the recent research on GPGPU is focusing on achieving higher performance with GPUs for lower grid sizes also. With evolution in technology it will surely be possible and worth replacing CPUs with GPUs for atleast a few specific computation domains in near future.

REFERENCES

[1] Kuen Hung Tsoi and Wayne Luk, Department of Computing, Imperial College, London, UK, ***"Axel: A Heterogeneous Cluster with FPGAs and GPUs,"*** khtsoi@doc.ic.ac.uk, wl@doc.ic.ac.uk.
[2]http://www.amax.com/hpc/productdetail.asp?product_id=superg

[3]http://www.nchpc.com/2010/02/appro-hyperpower-cluster-featuring.html
[4]http://aces.snu.ac.kr/Center_for_Manycore_Programming/SnuCL.html
[5]http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&prid=678988&releasejsp=release_157
[6] Volodymyr V. Kindratenko, Jeremy J. Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, Wen-mei Hwu, "GPU Clusters for High-Performance Computing"
[7] (2009) GraphStream, Inc. website. [Online]. Available: http://www.graphstream.com/.
[8] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Tureka, "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," *Parallel Computing*, vol. 33, pp. 685-699, Nov 2007.
[9] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W. Hwu, "QP: A Heterogeneous Multi-Accelerator Cluster," *in Proc. 10th LCI International Conference on High- Performance Clustered Computing*, 2009. [Online]. Available: http://www.ncsa.illinois.edu/~kindr/papers/lci09_paper.pdf
[10] (2008) Intel 64 Tesla Linux Cluster Lincoln webpage. [Online] Available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/
[11] (2009) Accelerator Cluster webpage. [Online]. Available: http://iacat.illinois.edu/resources/cluster/
[12] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, "GPU Computing"
[13]http://www.mathworks.in/company/newsletters/articles/gpu-programming-in-matlab.html
[14]http://www.sciencedirect.com/science/article/pii/S1093326310000914
[15] http://www.nvidia.com/object/nvidia-kepler.html
[16] http://en.wikipedia.org/wiki/CUDA
[17] https://www.mathworks.in/discovery/matlab-gpu.html
[18]http://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CC0QFjAA&url=http%3A%2F%2Fwww.cs.kent.edu%2F~zhao%2Fgpu%2Flectures%2Fcuda.ppt&ei=oWiYUY2kHMeJrQei-IHIBQ&usg=AFQjCNHiNjqvwFSmg7s8svbpRTyLPiz5cg&sig2=kmax6UVQf1aGhLVAzxVezA
[19] Jääskeläinen, Pekka O.; de La Lama, Carlos S.; Huerta, Pablo; Takala, Jarmo H. (July 2010). "OpenCL-based design methodology for application-specific processors". *2010 International Conference on Embedded Computer Systems (SAMOS)* (IEEE): 223–230. doi:10.1109/ICSAMOS.2010.5642061. ISBN 978-1-4244-7936-8. Retrieved 17 February 2011.
[20]http://www.altera.com/products/software/opencl/opencl-index.html
[21]http://www.eecg.toronto.edu/~jayar/fpga11/Singh_Altera_OpenCL_FPGA11.pdf
[22] http://wiki.tiker.net/CudaVsOpenCL
[23] http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf