iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 6 **Issue:** VI **Month of publication:** June 2018

DOI: http://doi.org/10.22214/ijraset.2018.6016

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue VI, June 2018- Available at www.ijraset.com

GPU Programming Using NVIDIA CUDA

Siddhante Nangla[1], Professor Chetna Achar[2]
[1, 2]*MET's Institute of Computer Science, Bandra Mumbai University*

Abstract: ***GPGPU or General-Purpose Computing on Graphics Programming Unit, also called as Heterogeneous Computing, is rapidly emerging into an area of great interest to computer scientists and engineers. A single or a small group of connected GPUs can solve certain classes of computational problems faster than a multicore CPU can. GPUs can effectively and efficiently solve problems that have a high level of parallelization in them. To achieve this sort of GPU programmability, a C-based framework called CUDA exists. CUDA (Compute Unified Device Architecture) is a framework created and maintained by NVIDIA to help simplify the task of GPU programming. This paper presents the foundations of this computational model that can be harnessed for solving complex computational problems, while also mentioning its limitations.***
Keywords: Block, Compute Unified Device Architecture, Device, General Purpose Computing on Graphics Programming Unit, Grid, Graphics Processing Unit, Host, Heterogeneous Programming, Heterogeneous Computing, Thread.

I. INTRODUCTION

A GPU is a small onboard (integrated) or dedicated chip that is used to, primarily, render images to a video monitor [1]. Since their advent, GPU's have been heavily used for gaming [1].
More generally, though, a GPU is a chip that can very easily and efficiently launch multiple threads and execute instructions in a parallel fashion. [2] Contrary to a CPU, a GPU contains several hundred cores that can handle several hundreds of simultaneously running threads [1]. What's more is that this execution model is more power and cost efficient than an equivalent CPU [1]. This large number of threads forms the base for parallel programming in CUDA and GPU accelerated computing at large.

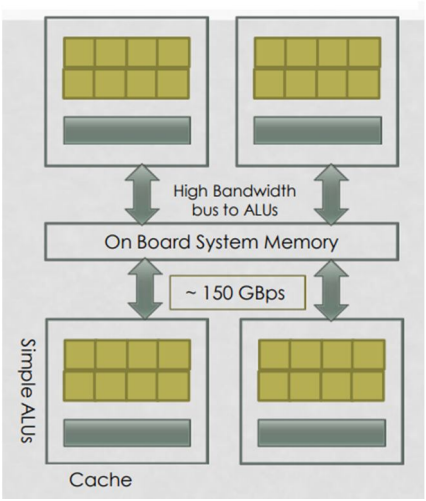


Fig. 1: A generic GPU architecture [5]

The large number of Arithmetic Logic Units is so designed that they can service and manage a very large number of threads simultaneously. [5] A GPU is a device that facilitates SIMD (Single Instruction, Multiple Data) operations due to its inherent multithreaded operation mode. [6] CUDA helps in harnessing that massive computational power for general computing.

II. LITERATURE REVIEW

NVIDIA CUDA is rapidly becoming the core for a lot of frameworks and libraries that need high performance. The CUDA framework speeds up AI libraries such as Tensorflow and OpenCV. Both are used in AI and machine learning to build models and analyse data. It is clear that, as machine learning and AI, as a whole, evolve, so too will the need for more speed and faster processing be required from the machines and processors that run those programs. CUDA is applied in diverse areas including finance, computer simulations and several other scientific research areas. Thus, the study of NVIDIA CUDA is critical if higher throughput and reduced latency is what is desired by programmer in applications they develop.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue VI, June 2018- Available at www.ijraset.com

III. MULTICORE CPUS VS. GPUS

CPUs like, say, the Intel Core 2 Duo are good at doing a few tasks concurrently (usually two or three) very quickly. Graphics cards or GPUs are good at doing a massive number of tasks at the same time and doing them very quickly. In order to accomplish this feat, graphics cards use hundreds of ALUs. Each of these ALUs can be programmed using CUDA. This enables programmers to harness a very large amount of computational power. However, this computational model does put certain limits on the types of applications that can be designed using CUDA.

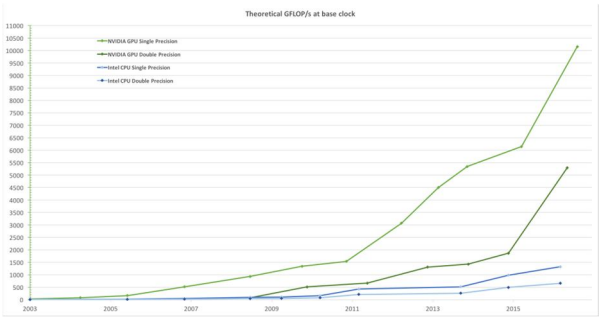


Fig. 2: Performance of CPU vs. GPU measured in GFLOPS over the years [11]

As can be seen from Fig. 2, a GPU offers far more performance than an equivalent CPU.
The performance of these devices is measured in GFLOPS or Giga Floating Operations per Second. FLOPS is a way of measuring the performance of computing systems that make use of floating point calculations in their operations. This performance difference is primarily because the GPU dedicates more of its transistors to data processing, rather than data storage, as in the case of CPU caches.

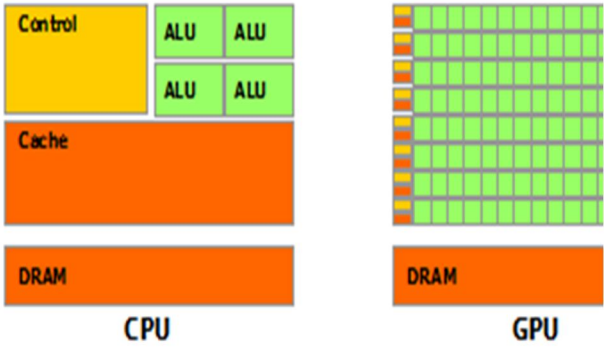


Fig. 3: CPU vs. GPU block diagram [11]

A. GPUs have a smaller cache, but many more ALUs compared to a general CPU. This is what gives them their speed. Many hundreds of ALUs run in parallel on the same instruction and produce results.
B. Most CUDA programs have high arithmetic intensity, which means that they are very calculation intensive.
C. The lower amount of cache in GPUs is more than compensated by computational speed of its several cores. This is because a GPU has a lot more ALUs than a multicore CPU.

In CUDA, a single program is written and that is executed across all the cores of the GPU. Due to this, there is a lessened requirement for advanced flow control, as is usually seen in CPUs [11]

IV. CUDA

At its core, CUDA (Compute United Device Architecture) is parallel programming framework and platform [3], that enables a developer to harness the features of an NVIDIA CUDA enabled GPU to create parallel programs that can run on any NVIDIA CUDA enabled GPU.
The real limiting factor here is that not all applications can be parallelized. CUDA is primarily a C-Based API [3], but has support for other languages including FORTRAN, C++ [3] and Python. [4]

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue VI, June 2018- Available at www.ijraset.com

V. THE CUDA PROGRAMMING MODEL

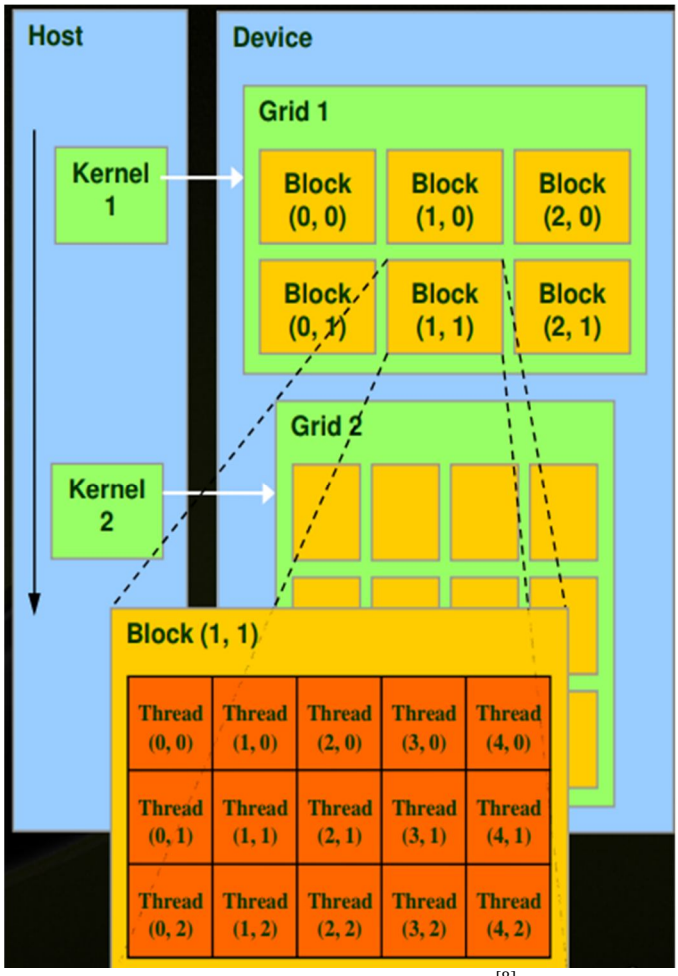


Fig. 4: CUDA Memory Setup [8]

CUDA applications run natively on an NVIDIA enabled GPU.
While they can run on a CPU also, they will never be as fast as a CUDA program running on a native GPU.
A CUDA program is divided into two parts: [8]
As Host and Device maintain separate memories for execution, the host has the main responsibility of moving the data from the CPU memory (Host memory) over to the GPU memory (Device Memory).
The host is in control of calling the device code when needed.
A kernel is a piece of code that is concurrently executed on the Device. [7]
Multiple kernels may be present in the same CUDA application; however, the CUDA runtime will schedule only one kernel to run at a time on the Device. [7]
The defining characteristic of a kernel is that it's written as a serial program, that is then run on multiple threads on the GPU

VI. CUDA COMPUTE CAPABILITY

The compute capability of a device is represented by a version number. It helps to decide which features are supported by a particular GPU and allows for the correct feature set to be used by a CUDA application when running.

VII. CUDA GRIDS, BLOCKS AND THREADS

A. Kernel functions (Kernels) are executed on the device in grids of threads. [9]
B. A single grid contains a block of threads.
C. A single block contains a group of threads. The maximum number of threads in a group is 1024 for CUDA Compute Capability 2.0. [9]

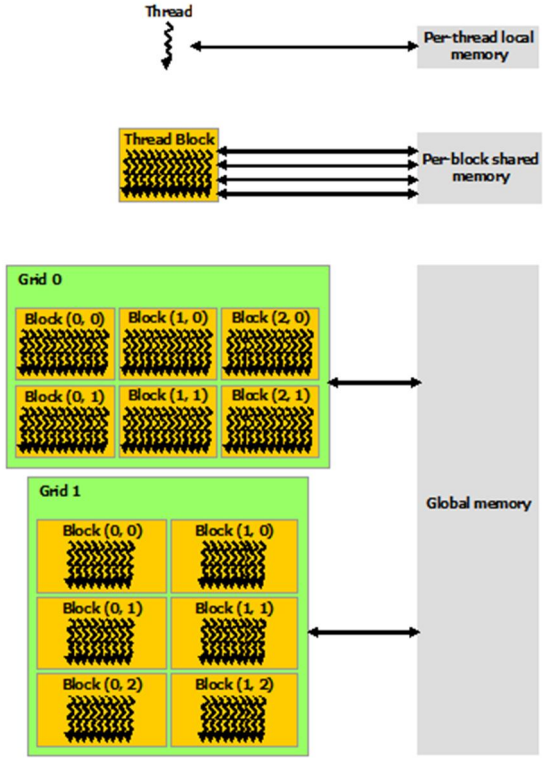


Fig. 5: Memory Hierarchy in CUDA. [10]

Technical Specifications	1.0	1.1	1.2	1.3	2.0
Number of 32-bit registers per MP	8 K		16 K		32 K
Maximum amount of shared memory per MP	16 KB				48 KB
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				

Fig. 6: Memory Specification for different compute capabilities. The technical specification specifies the CUDA compute capability

VIII. CUDA MEMORY AND ITS HIERARCHY

Three levels of memory exist

A. Local Thread Memory

This is memory that's local to a thread. It cannot be shared between threads in the same block or across blocks. Synchronization of this memory is not needed. [12] This is mainly because the memory is local to the thread and race conditions are not defined as the data cannot be shared.

B. Shared Memory

This is memory that is shared between all threads running in the same block. It has a slightly higher access time than local thread memory, but is faster than accessing Global memory

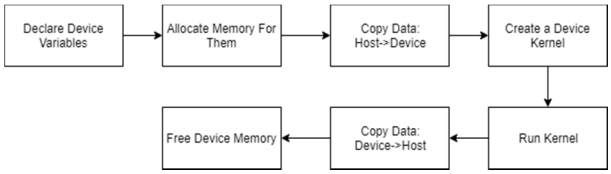
It can be accessed nearly 100 times faster than global memory. [12] Each block has a limited amount of memory and it can be accessed from the kernel. [12] Shared variables are declared inside the kernel using the __shared__ keyword. This memory needs to be synchronized to avoid deadlocks. It is achieved by using the __syncthreads() barrier function inside a kernel. [12]

C. Global Memory

This is the slowest level of memory access. Grids can share data amongst themselves here. All data here is available to all threads running across grids. It has latency around 100 times higher than shared memory. [12] These variables are declared in the kernel using the __device__ keyword.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 6.887
Volume 6 Issue VI, June 2018- Available at www.ijraset.com

IX. STEPS IN THE CREATION OF A CUDA PROGRAM



A. The following are the Basic Steps Involved in the Creation of a Cuda Program.

a) Declare the device variables that will be used. They can be shared, local or global variables.

b) Memory for the created device variables needs to be allocated on the GPU. This is achieved using the cudaMalloc(Device_Var, Var_Size) function.

c) The data in the host (CPU) Memory needs to be copied over to the device memory. This is achieved using the cudaMemcpy(Device_Variable, Host_Variable, SizeOfMemory,DirectionofTransfer) function.

d) Create a kernel to process the data. That kernel should be launched with an appropriate number of blocks and threads per block.

e) After the kernel has finished running and the data is ready on the GPU, it should be copied back to the host (CPU) memory. Again, this is achieved using This is achieved using the cudaMemcpy(Device_Variable, Host_Variable, SizeOfMemory,DirectionofTransfer) function.

f) Finally, allocated resources should be freed on the device using cudaFree(Device_Variable).

IX. CUDA CODE SAMPLE

The code that follows is a small comparison between CPU sequential code and an equivalent simple kernel. Both transform an image from the RGBA color space to the grey scale color space. The equation that is used for this purpose is .299f * rgba.x + .587f * rgba.y + .114f * rgba.z; where the rgba pointer is a pointer to the image in memory. The x, y and z elements are used to access the pixels' r, g and b components respectively in CUDA.

The CPU side of the code uses two for loops: One to iterate over the rows, while the other iterates over the columns. In that way, the code visits each pixel in the image sequentially once and performs the transformation, storing the result in an array.

The kernel, on the other hand, uses no such loop. Each individual thread of the GPU acts on one pixel of the image and converts that pixel to a greyscale version. This example alone demonstrates the power of heterogeneous computing for more complex applications. The equation (blockIdx.x * blockDim.x) + threadIdx.x is used to compute a unique thread index in device memory and helps in mapping image pixels to threads in a one to one fashion. That is, one pixel will be operated upon by one device thread. Finally, the kernel is called from the host side by using the <<< x, y>>> operator. Here, x specifies the number of blocks and y, the number of threads in that block.

X. APPLICATIONS

A. These Include Applications such as: [13]

1) AI Applications:

a) Solving Deep learning models.

b) Running computer vision algorithms in real time

2) Image Processing

a) Convert Images in Real Time

b) Perform real time video analysis

3) Searching and sorting

a) Parallel versions of most sorting and searching functions exist

b) Search: depth-first search, breadth-first search, finding all nodes within one connected component

c) Sorting: Quick-sort

4) Weather monitoring

a) Creating and running weather models.

5) NBody methods:

a) Astronomy (formation of galaxies)

b) Chemistry: Molecular Dynamics , molecular modelling

c) Physics: Fluid Dynamics, Plasma Physics

6) Graph Traversal

a) Serialization/Deserialization
b) Maze generation
c) Collision detection

XI. LIMITATIONS

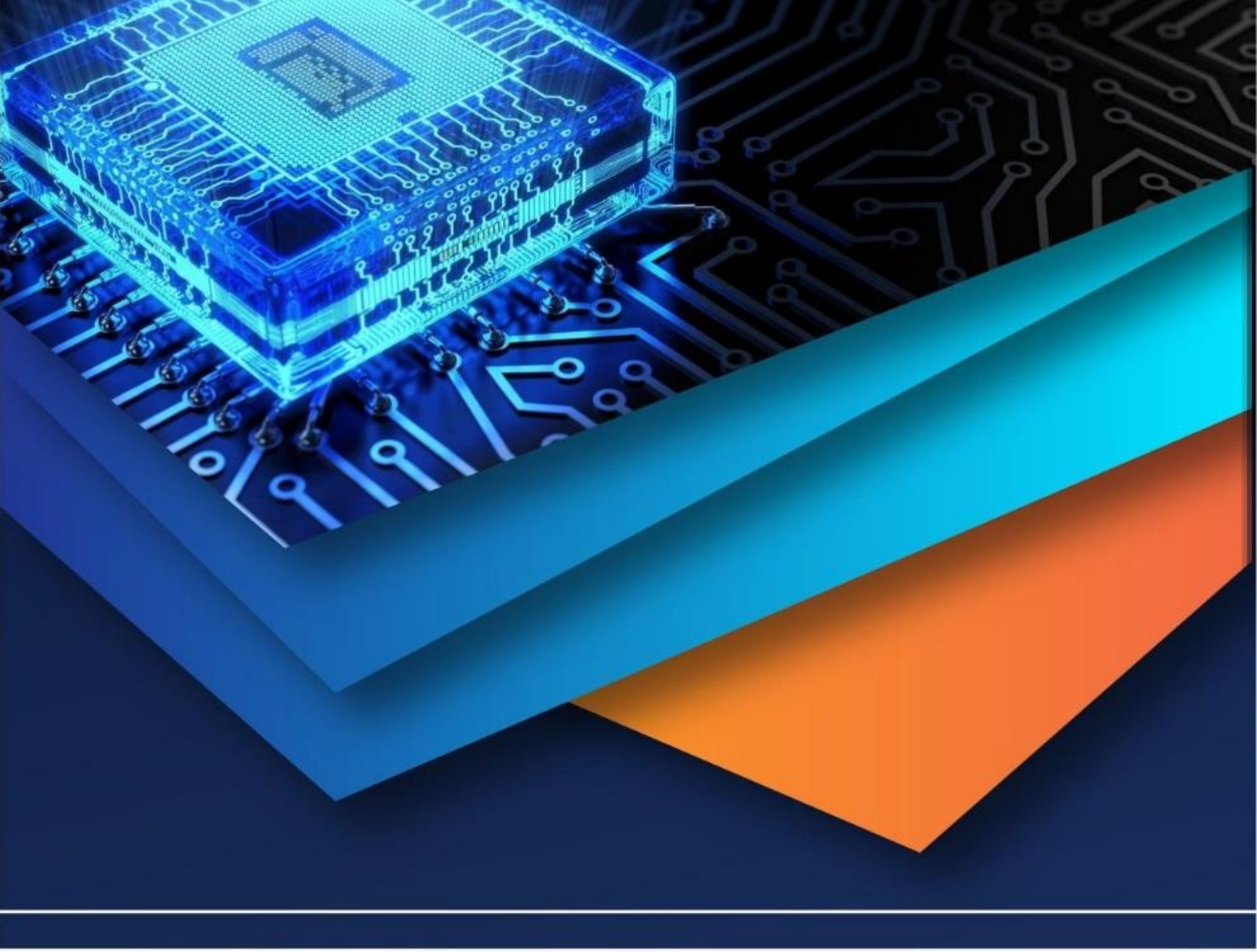
A. CUDA and GPGPU applications at large are good at solving problems that involve heavy number crunching.
B. As such they will not be useful for every computing problem.
C. The algorithm in question has to have a good level of parallelism for it to map to the multi threaded framework the GPU work with.
D. Not all CPU (sequential) code can be converted into GPU (parallel) code
E. Parallelising a serial implementation may sometimes be very complicated.

XII. CONCLUSION

This paper introduced the basic concept of Heterogeneous programming in the context of using CUDA to develop high performance applications that run on a GPU instead of a CPU. It explored the memory model of the GPU and explained the fundamental differences between a CPU and a GPU. It also explained what CUDA was and how it fit in heterogeneous computing and the steps necessary to run a CUDA program. This paper also introduced a small CUDA reference kernel to show how a simple kernel program might be written. Finally, it explored some applications and limitations of Heterogeneous Programming.

REFERENCES

[1] https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu
[2] http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf Chapter 2
[3] https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/
[4] https://developer.nvidia.com/how-to-cuda-python
[5] https://www.cs.indiana.edu/~achauhan/Teaching/B649/2011-Fall/StudentPresns/gpu-arch.pdf
[6] http://www.oxford-man.ox.ac.uk/gpuss/simd.html
[7] http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
[8] http://www.serc.iisc.in/facilities/wp-content/uploads/2015/02/Introduction_to_CUDA.pdf
[9] https://www.3dgep.com/cuda-thread-execution-model/
[10] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy__memory-hierarchy-figure
[11] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction
[12] https://www.3dgep.com/cuda-memory-model/
[13] https://streamhpc.com/blog/2013-06-03/the-application-areas-opencl-and-cuda-can-be-used/



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY