A Deep Dive into CUDA Architecture

[1]Dr. Paresh M Dholakia, [2] Pankti Dholakia

[1]Associate Professor, [2]Student
[1]Department of Electronics and Communication Engineering,
[1]VVP Engineering College, Rajkot, India

Abstract : Over the last decade, GPUs have been extensively used to meet the computational needs for big data and machine learning applications. In this study we propose to study and review NVIDIA's CUDA (Compute Unified Device Architecture) – a parallel computing library that provides APIs that allows varied applications to utilize the underlying GPU. This paper will discuss the CUDA model and improvements since CUDA 4.0 and how it compares to OpenCL. This graphic library enables compute intensive applications to leverage GPUs for parallel processing and this research explores the dominant performance of CUDA with GPUs using exploratory deep learning applications**.**

IndexTerms **– GPU, NVIDIA, CUDA**

I. INTRODUCTION

As multicore CPUs and GPUs have evolved, processor chips have transformed into parallel systems. The graphics processing units (GPUs) from NVIDIA are extremely potent and parallelized. Due to their extensive computational capability, GPUs outperform CPUs by a wide margin [1]. This is because they have many more processor cores than the CPU and can handle thousands of concurrent threads on each core. Therefore, it is imperative that there was need of software to be designed and developed to take advantage of multithreading, where each thread runs parallelly on a processor, resulting in a huge increase in program speed. A parallel processing model that allows a parallel multicore programming environment is needed to create such scalable parallel applications. CUDA was introduced as "Compute Unified Device Architecture," a general-purpose parallel computing platform and programming style, to handle many difficult computational tasks more quickly than on a CPU

II. GPU

A central processing unit (CPU) is designed to execute complicated functions, including security, complex control flows, virtual machine emulation, time slicing, etc. Graphical processing units (GPUs), in contrast, are good at just one thing. They execute billions of low-level repetitive jobs. They feature hundreds of arithmetic units (ALUs), as opposed to regular CPUs, which typically only have 4 or 8. They were initially created for the depiction of triangles in 3D graphics [5].
Many kinds of scientific algorithms spend most of their time executing billions of repeated arithmetic operations, which is exactly what GPUs are brilliant at. The strength of GPUs has been quickly tapped by computer scientists for computational scientific applications. They use multi-core chip SIMD execution within a single core (many execution units performing the same instruction) and multi-threaded execution on a single core (multiple threads executed concurrently by a core).

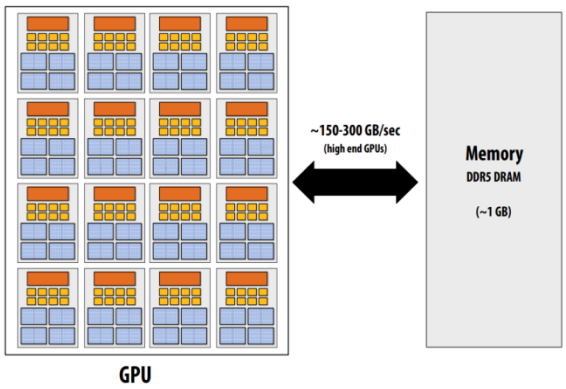

Figure 1. GPU basic architecture

They were initially only used for graphics but now, a range of general-purpose, non-graphic applications are increasingly using GPUs. Examples of applications include computational chemistry, sorting, sparse matrix solvers, and physics models.
The GPGPU (General Purpose GPU) programs execute on the multi processors employing a lot of threads at once. These applications are hence quite quick.

III. CUDA- AN NTRODUCTION

The framework for all-purpose processing on Nvidia GPUs is called Compute Unified Architecture (CUDA). Using CUDA, operations that do not need sequential execution may be carried out concurrently on a GPU. It is quite simple to offload computation-intensive jobs to Nvidia's GPU using CUDA and it initially supported the languages C, C++, and Fortran [1]. When parallelization is possible and high performance is required as well as in domains that call for a lot of processing capacity, CUDA is employed. Simply put, CUDA is the software and GPU is the hardware. Since an Nvidia GPU is necessary to run CUDA, it can be downloaded for free from the Nvidia website.

Only NVIDIA GPUs built on the Tesla architecture support CUDA. Tesla, Quadro, and GeForce 8-series graphics devices all support CUDA. These graphics cards are simple to utilize in servers, laptops, and PCs.

.

IV. HISTORY

Nearly 20 years ago, parallel computing first used GPU technology. A platform for general-purpose programming models called Brook was unveiled by a team of Stanford academics. Nvidia provided funding for the study, and the principal investigator, Ian Buck, eventually joined the company to work on the development of CUDA, a for-profit solution for GPU-based parallel computing. Nvidia has created 32 versions altogether thus far.

NVIDIA Tesla architecture was used to launch CUDA in 2007. The language used to write programs that use the compute-mode hardware interface on GPUs is like C language. CUDA's abstractions closely resemble the capabilities and performance traits of contemporary GPUs (design goal: maintain low abstraction distance). It is relatively a low-level language Machine learning, physics research, medical science analysis, supercomputing, cryptocurrency mining, and scientific modelling and simulations are just a few of the industries that employ CUDA.

V. THE CUDA SYSTEM

To make use of the GPU's tremendous parallelism, NVIDIA created the unique C-based language CUDA. CUDA has a unique C function called kernel, which is just C code that runs on graphics cards simultaneously on a set number of threads. In CUDA, thread definitions are made using a grid structure. The CUDA programming paradigm combines parallel and serial executions and they are exposed to the programmer as a minimal set of language extensions.

5.1 Architecture

Simple C code is executed serially on the host CPU. The kernel function, which is performed on several threads concurrently on the GPU (also known as the device), describes parallel execution. The kernel code is written in C and supports just one thread. When this function is invoked, the explicit numbers of thread blocks and the number of threads within those blocks that run this kernel in parallel are provided. There are two types of variables - Grid and block that are expressed in three angle brackets (<<<grid, block >>>) in the kernel function call. Grid and thread blocks are constructed dynamically in this invocation. The allowable sizes, which are listed in the following section, must be smaller than the value of the grid and block variables. Hardware rather than software schedules the threads. Kernel functions always have a void return type. The qualifier __global__ indicates that it is a kernel function that will be run on the GPU.

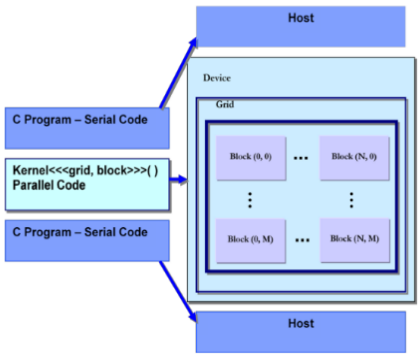

Figure 2. Heterogeneous CUDA Architecture [3].

At the heart of the CUDA parallel computing architecture are three fundamental abstractions:

- a hierarchy of thread groups
- shared memories
- barrier synchronization

The multithreaded Streaming Multiprocessors (SM) that make up the core of the CUDA architecture are scalable. A group of execution units, a group of registers, and a section of shared memory are present in every SM.

Figure 3. Streaming Processors in CUDA.

5.2 Grid and Block Model

Several threads running on a single processor make up a thread block. One-dimensional, two-dimensional, or three-dimensional thread blocks make up the Grid. Each of these blocks are further broken down into 1-D or 2-D threads. The warp is the fundamental unit of execution in an NVIDIA GPU. The threads are arranged in warps within a thread block. In a warp, 32 threads are typically clustered together. A warp's threads are scheduled for execution all at once. Multiple warps can run on an SM at the same instance. Since shared memory allows for communication between all threads in a single thread block, they are all run on the same multiprocessor. These threads can be synchronized in this manner.

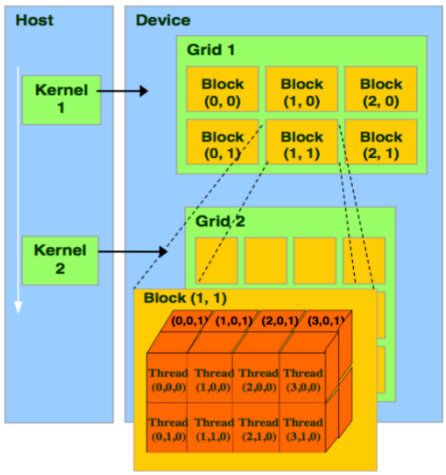

Figure 4. Grid and Block Structures.

The CUDA paradigm offers certain built-in variables to make effective use of this structure. While *gridDim* provides the dimensions of the grid, *blockIdx* (values from 0 to gridDim-1) is used to access a thread block's id and *blockDim* (values from 0 to gridDim-1) to access a thread block's dimension. The *threadIdx* variable, which accepts values between 0 to blockDim-1, uniquely identifies each thread. In the threads, *WarpSize* determines the warp size. These variables are all included in the kernel. The largest size for each grid dimension is 65535, while a thread block's x, y, and z dimensions are 512, 512, and 64, respectively. The number of thread blocks that are assigned to each multiprocessor depends on how much shared memory and register space each thread block requires

5.3 Memory Model

There are different memory areas for the CPU and GPU. This implies that before a computation can begin, data that the GPU will process must be sent from the CPU to the GPU, and when processing is complete, the data must be transferred back to the CPU [1]. All threads and the host (CPU) have access to the global memory. The CPU allocates and deallocates global memory, which is used to set up the data that the GPU will operate with.

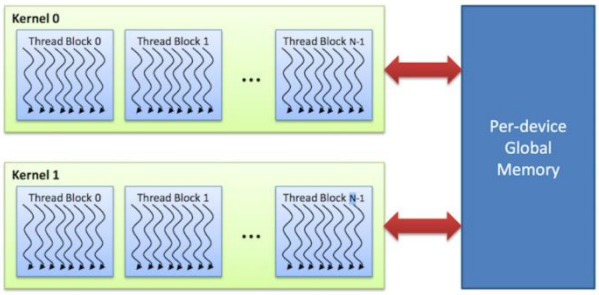

Figure 5. Global Memory Representation

Additionally, each thread block has its own shared memory. It is substantially quicker than the global memory and is exclusively accessible by threads inside the block. In contrast to device memory, shared memory is "local" to each multiprocessor and enables more effective local synchronization. To achieve optimal performance, it requires careful management and is only present for the duration of the block. It is also known as parallel data cache (PDC). It is broken up into several pieces. The shared memory that each thread block in a multiprocessor access is private to that thread block and is not accessible to any other thread blocks in that multiprocessor or any other multiprocessor.

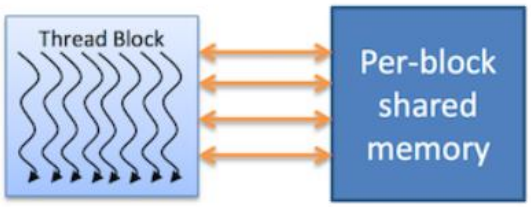

Figure 6. Shared Memory Representation

Additionally, each thread has a private local memory that is managed automatically by the compiler and exists only while the thread is active.

Figure 7. Local Thread Memory Representation

5.3 Synchronization and Processing Flow

5.3.1 Synchronization

The hardware thread-barrier method *syncthreads*() serves as a synchronization point and is provided by the CUDA API for synchronization purposes among threads [1]. This function is implemented in hardware since threads are scheduled in hardware. Until all of the threads have arrived to the synchronization point, the threads will wait there. Per-block shared memory makes it feasible for thread communication, if necessary. Therefore, only at the thread block level can threads be synchronized. The number of threads per block and the number of thread blocks per grid should be carefully assigned to maximize the use of the available resources. Less threads per block result in slower load times when reading from device memory, and one block per multiprocessor causes the multiprocessor to become inactive while thread synchronization is taking place. Since there are more multiprocessors in the device than blocks, there should be at least two times as many blocks. Additionally, because it reduces the number of underpopulated wraps, allocate the thread count per block as multiples of the warp size.

5.3.2 Process Flow

Memory must be allocated on the device prior to calling the kernel function, and if the kernel function must work with some data, the data must be transferred from the CPU memory to the GPU device memory. Device memory can be allocated as CUDA arrays or as linear memory. When a variable begins with the qualifier __device__, it means that memory on the device has been set aside for it.

Device memory can be allocated and released at runtime using API calls like *cudaMalloc*() and *cudaFree*() [1,2]. Like this, in order to obtain results from a kernel function, data from device memory must be transferred back to host memory. The CUDA offers methods like *cudaMemCpyToSymbol*(), *cudaMemCpyFromSymbol*(), *cudaMemCpy*(), etc [1]. to copy data to and from the device to the host. The control flow is carried out in the following steps:

1. Separately allocate memory on the host and the device. The CPU can read and write to GPU memory using the memory copy methods.
2. If necessary, copy data from the host to the device using the CUDA API.
3. Parallel kernel operations are carried out on each core.

4. Using the CUDA API, copy data back from the device to the host.

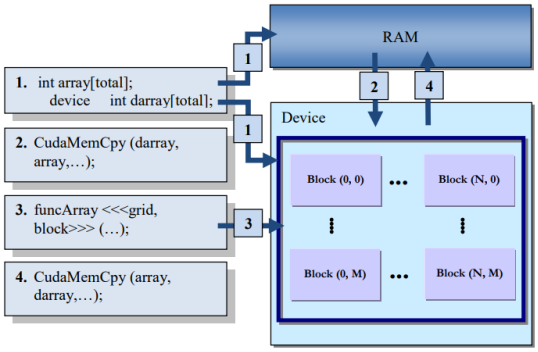

Figure 8. Processing Flow [3].

VI. CUDA 4.0

Since the first CUDA toolkit, the 4th edition introduced considerable changes for broader developer adoption. The first version had been developed keeping researchers and early adopters in consideration. The following versions introduced changes for data and machine learning scientists and application innovation leader. In CUDA 4.0 significant changes were made to make it suitable for broader developer adoption outside of specialist developers and enabled broad spectrum of development.

6.1 Easier Application Porting

GPUs might now be shared among several threads. Pthreads and OpenMP threads can share a GPU for easier porting of multi-threaded applications [2]. Enabled starting several kernels simultaneously from various host threads. Created new and straightforward context management APIs that eliminated context switching cost while still supporting the older context migration APIs. All the system's GPUs were accessible to each host thread. The restriction of single thread per GPU was eliminated. They made it simpler for apps to utilize many GPUs, and single-threaded applications began to gain from this.

6.2 NVIDIA GPUDirect 2.0 Protocol

For third-party devices, direct access to GPU RAM was made available. By Mellanox and Qlogic, unnecessary sys mem copies and CPU overhead were removed. The performance in communicating improved by 30% [2]. Memory transfers, synchronization, and peer-to-peer access were added. There was an increase in programmer productivity and less code.

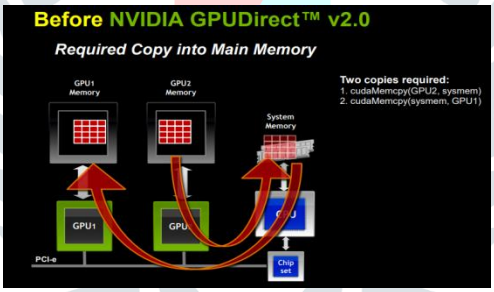

Figure 9. CPU Overhead before NVIDIA GPUDirect [2].

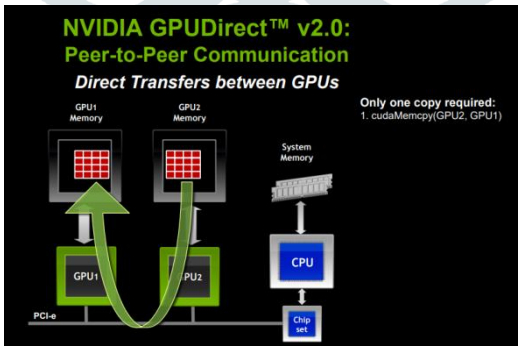

Figure 10. P2P Communication – NVIDIA GPUDirect 2.0 [2].

6.3 Unified Virtual Addressing

Pointers may be accessible from GPU code regardless of where they are in the system, whether they are in device memory (on the same or a separate GPU), host memory, or on-chip shared memory, thanks to a technique called unified virtual addressing, or UVA. Additionally, it enables the usage of *cudaMemcpy()* without providing the precise locations of the input and output parameters. Zero-Copy memory, which is mounted host memory directly accessible by device code through PCI-Express, without a *memcpy*, is made possible by UVA. Because it is constantly accessible using PCI-Express, Zero-Copy offers some of the performance of Unified Memory but none of its simplicity.

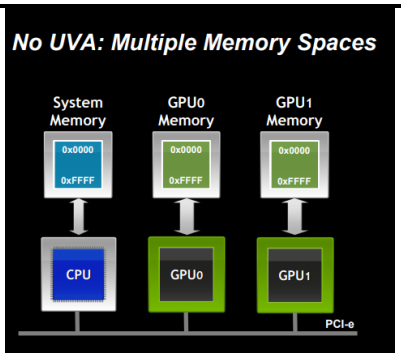

Figure 11. Without UVA Memory Access [2].

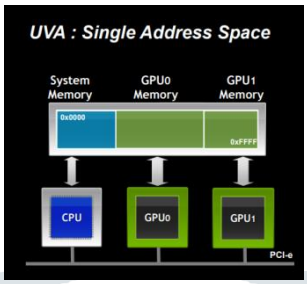

Figure 12. UVA Based Memory Access [2].

Unlike Unified Memory, UVA does not automatically transfer data across physical locations. It would take a lot of engineered efforts to create Unified Memory since it can automatically move data at the level of individual pages between host and device memory. This would need new capability in the CUDA runtime, the device driver, and even the OS kernel.

VII. OPENCL

In order to create a standard for heterogeneous computing that was not limited to solely NVIDIA GPUs, Apple and the Khronos organization introduced OpenCL, an abbreviation for the Open Computing Language. For GPU programming, OpenCL offers a modular language that works with CPUs, GPUs, DSPs, and other types of processors. With the aid of this portable language, it is possible to create programs or applications that are both adaptive and broad enough to work on a wide range of hardware platforms and architectural styles. Programs written in OpenCL are portable, vendor and device-independent, and may be accelerated on a variety of different hardware platforms. The OpenCL C language is a constrained variant of C99 that incorporates enhancements suitable for running data-parallel algorithms on diverse hardware [5].

VIII. CUDA V/S OPENCL

For GPU programming, OpenCL guarantees a portable language that is proficient at working for extremely unconnected parallel processing machines. Since most have highly distinct feature sets, this in no way implies that a code will work on all devices, if at all. It takes some extra work to make the code cross-platform while avoiding vendor-specific extensions. An OpenCL kernel may be built during runtime, unlike the CUDA kernel, which would increase an OpenCL's execution time. On the other hand, the compiler might be able to produce code that makes greater use of the target GPU thanks to this just-in-time compilation.

Since only NVIDIA GPUs support CUDA, but OpenCL is an open industry standard that supports NVIDIA, AMD, Intel, and other hardware devices, this is perhaps the most noticeable distinction between the two [4]. On the other hand, CUDA does not offer CPU fallback, forcing developers to include if-statements in their code that assist to distinguish between the existence of a GPU device at runtime or its absence. Additionally, OpenCL offers CPU fallback, making code maintenance easier.

CUDA can run on Windows, Linux, and MacOS but only if used with NVIDIA hardware. But practically every operating system and most hardware types can run OpenCL [4]. The hardware is still the key deciding factor in the OS support comparison, as OpenCL is compatible with practically all operating systems whereas CUDA is only compatible with the top operating systems.

Another notable distinction between CUDA and OpenCL is that OpenCL is an open-source framework, whereas CUDA is an NVIDIA-exclusive framework. The overall better option depends on the app you want. This difference has its own advantages and disadvantages. In general, choosing CUDA is the best option if the software of your choosing supports both CUDA and OpenCL because it produces higher performance results in this situation. This is due to the excellent assistance offered by NVIDIA. A current NVIDIA card can let you get the most out of CUDA-enabled programs while having decent compatibility in non-CUDA apps if some apps are CUDA-based and others use OpenCL [4].

Since libraries provide access to a set of functions that have already been optimized to take advantage of data parallelism, they are essential to GPU computing. Due to its support for templates and free raw math libraries that provide high speed math routines, CUDA performs exceptionally well in this category with libraries like -

- Complete BLAS Library is cuBLAS.
- Random Number Generation (RNG) Library, cuRAND
- Sparse Matrix Library: SPARSE
- Fast Fourier Transforms used in NPP, or Performance Primitives for Image & Video Processing.
- Templated Parallel Algorithms and Data Structures: Library Thrust
- h - floating-point C99 Library

Alternatives to OpenCL may be constructed quickly and have improved recently, but none compare to the CUDA packages.

IV. SUMMARY

Compared to conventional general-purpose computing on GPUs (GPGPU) utilizing graphics APIs, CUDA provides several benefits. It allows programs to read from arbitrary locations in memory using scattered reads. Unified virtual memory (CUDA 4.0 and higher) and Unified memory are features of the CUDA architecture (CUDA 6.0 and above) [5]. The quick shared memory area that is made available by CUDA and may be shared by several threads. In contrast to texture lookups, this may be utilized as a user-managed cache to provide better bandwidth and quicker readbacks and downloads to and from the GPU. Over time, CUDA has advanced and expanded in breadth, mostly in tandem with the advancement of NVIDIA GPUs. Many applications that require high floating-point computation performance now use CUDA and NVIDIA GPUs.

REFERENCES

[1] NVIDIA CORPORATION, CUDA Reference Manual, http://developer.nvidia.com/cuda

[2] NVIDIA CORPORATION, CUDA Toolkit 4.0 Overview Guide, CUDA_Toolkit_4.0_Overview.pdf

[3] Inam, Rafia, A* Algorithm for Multi-core Graphics processors, Master's Thesis, Chalmers University of Technology, Göteborg, 2010.

[4] Zunitch, Peter (2018-01-24). "CUDA vs. OpenCL vs. OpenGL". *Videomaker*. Retrieved 2018-09-16.

[5] Manavski, S.A., Valle, G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics 9 (Suppl 2), S10 (2008).