Sindh Univ. Res. Jour. (Sci. Ser.) Vol. 50 (3D) 01-05 (2018)

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES)

CUDA: A new paradigm for parallelization and computational efficiency

J. DEVI, J. KUMAR, S. PARVEEN

Department of Information Technology, Quaid-E-Awam University of Engineering Science and Technology, Nawabshah , Sindh Pakistan

Received 10th June 2018 and Revised 15th September 2018

Abstract: This paper will present a comprehensive evaluation of the CUDA programming model and discuss it efficacy with regards to the parallelization of the highly resource-intensive computational processes. The paper also presents a comprehensive discussion on the architecture of the CUDA programming model and its utilization for the purpose of digital image processing.

Keyword: CUDA programming model and discuss

1. INTRODUCTION

CUDA or Compute Unified Device Architecture is essentially a sophisticated and state-of-the-art programming model and computing platform with parallel processing capabilities. CUDA allows smooth execution and implementation of parallel programming codes like compilation of a simple C++ algorithm and allows conceptualization and development of systems that can run on a wide range of devices such as laptops, embedded systems, processor clusters and even smart phones and tablets. For execution, modification and development of CUDA programming codes, programmers can utilize the standard C software development tools facilitating easy usage. In order to properly appreciate the significance of CUDA programming architecture it is prudent that there is a brief discussion on the concept of parallel programing, especially due to the fact it significantly improves algorithm performance and decrease latency (Cheng, *et. al,* 2014).

In the recent times the concept of parallel computing has been the subject of focus primarily due to the prospect of speeding up the computational process and improving performance. When seen in context of computational calculations, the processes are carried out in tandem across multiple processors. The inherent logic is to simplify the problem by breaking down a big task into small parts and solving those separately using different processors. The fundamental constituents of a parallel computing architecture are its hardware and software components and it is necessary that there is a cohesive relationship between the two. From a technical perspective, the computing architecture represents the hardware component and the parallel programming aspect represents the software component. In the present times, the concept of parallel processing is almost becoming standard in the sense that it drives or influences the process of computer architecture development (De Donno, *et. al,* 2010). Parallel processing can be described in terms of the task and the data that is being processed. Parallel processing of the task, indicate processing of multiple tasks in a parallel manner concurrently across multiple processor cores. Similarly, parallelism of data means computational processing of multiple data types at any given point of time. When seen in context of CUDA programming, the inherent logic is more suited towards data parallelism (De Donno, *et. al,* 2010).

2. EASE OF THE CUDA MODEL AND PROGRAMMING ARCHITECTURE

The inherent model of any programming logic serves as an abstraction layer that acts as a mediator between computer applications and their hardware mediated execution. **(Fig. 1)** below nicely illustrates the abstraction layer between the programs and their hardware execution (Cheng, *et. al,* 2014).

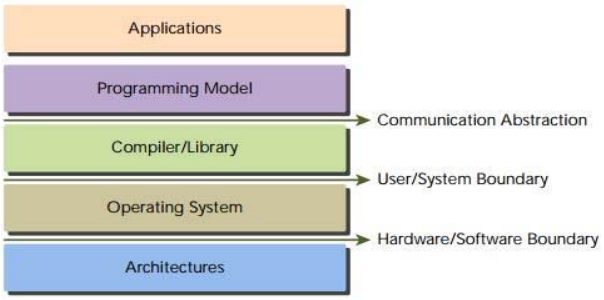

Fig.1. Abstraction between hardware and programs

The abstraction becomes possible through the use of a software compiler along with the hardware and a dedicated operating system. The software program is developed in a manner that there is coherence between its multiple components with regards to data processing and sharing and the manner in which the processes are

++ Corresponding author: jhernadevi@hotmail.com, jagdesh.k24@gmail.com, engr_sajida@hotmail.com

carried forward. While there could be multiple abstraction layers in any given parallel computing models, the CUDA model paradigm offer solutions or functionalities that facilitate optimum utilization of Graphics processing unit (GPU) processing capabilities. Graphics processing units or GPUs are known for their computational resourcefulness with regards to performance and speed. Originally developed for use in video cards, the graphics processing units integrate multiple processor cores in their architecture. This architectural characteristic allows processing of multiple threads concurrently with high performance and minimal software latency. The CUDA programming model allows optimal exploitation of GPU computational resources through a user-friendly and familiar programming environment such as C (Sharp, *et. al,* 2007). Software programmers can write simple C codes to implement the CUDA programming model and invoke parallel processes in millions concurrently (Vetter, *et. al,* 2007), (Gu, *et. al,* 2006).

The CUDA programming model allows complete unlocking of the parallel programming capabilities of the GPU based graphic cards through the use of customized programming language called CUDA C. Programs written in CUDA C can be seamlessly executed on any Graphics processing units that is compatible with the CUDA paradigm (Abecassis, *et. al,* 2015). It has already been mentioned that the strength of CUDA with regards to programming efficiency lies in the possibility of parallelization. In order to exploit the advantages of the CUDA programming model it is necessary to segment the executed algorithm in small processes with very little dependencies. Once this is done, the segmented tasks are mapped to separate process threads that are simultaneously executed on multiple processor cores of the Graphics processing units. The CUDA paradigm facilitates formation of units called wraps that consist of 32 different process threads. Wraps that are homogeneous or similar in nature are executed together in parallel and the efficiency increases further if the 32 process threads have similar instructions and dependencies. The wraps are further grouped into blocks that come together to form program grids. The parallel processing model is illustrated in **(Fig. 2)** below (Abecassis, *et. al,* 2015).

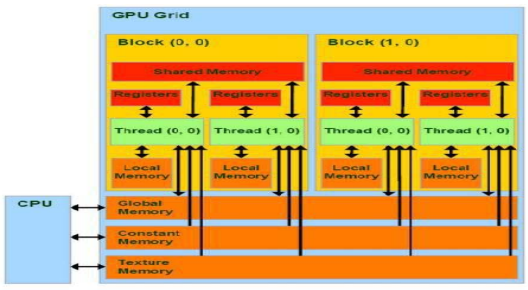

Fig.2. CUDA programming architecture

One of the most fundamental unit of the CUDA programming paradigm is the kernel which is essentially a program that is executed on the Graphics processing unit of the hardware. The Kernel can also be described as a program code that runs in a sequential manner and is managed by the CUDA programing model with regards to its segmentation and assignment into multiple GPU process threads. During the execution of CUDA, the primary CPU of the computing architecture acts as a main host and facilitates initiation of a single kernel at a time. For every kernel there are multiple processes threads that are executed concurrently on different processor cores. It has already been mentioned that homogeneous threads are grouped into wraps and it is important that all the threads have similar instructions. This is because for every wrap a single common instruction is executed at a single time and if instructions are different for different threads the efficiency is lost. Furthermore, individual threads within a wrap may conditions programming conditions. In such cases the execution process is diverted towards the condition branches till the end before they return to the main branch. This can also reduce the efficiency that is expected from the parallelization of the threads. In this context, non-inclusion of branch paths during the design process of the kernels is a good programming optimization strategy (De Donno, *et. al,* 2010).

Another advantage of the CUDA programming model lies in the fact that it allows execution of applications on computing systems that are highly heterogeneous in nature. This is possible through simple annotation of the programming code in the form of extensions towards the C programming convention. In a typical heterogeneous computing architecture, there are highly competent Graphics processing units that are integrated with the system CPUs. Each CPU and GPU units come with their own memory and the architecture can be distinguished into the following:

- The host that comes with its own CPU and memory unit called the host memory.
- The device which is essentially the Graphics processing unit that comes with its own memory unit called the device memory.

The host unit of the architecture is entirely independent from the device unit with regards to the execution of majority of the processes. Once a kernel process is initiated, the inherent control of the same lies with the host and this approach allows utilization of the CPU resources for carrying out other ancillary processes. The CPU is further assisted with the execution of the ancillary processes by data parallelization algorithms that are being executed on the system. Another noteworthy aspect of the CUDA paradigm is that it allows overlapping of GPU

processes with the host-device interaction or data exchange processes. An introspection of a standard CUDA code reveals that it is composed of a primary serial code that is supported by optimizing parallel code. **(Fig. 3)** below nicely illustrates the execution protocol of the serial code and the parallel code in the host and the GPU unit respectively (Cheng, *et. al,* 2014).

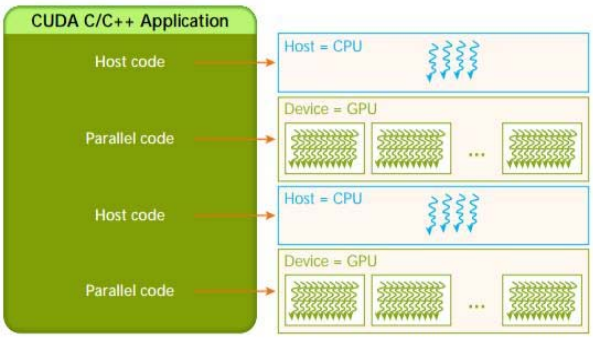

Fig.3. Execution of the serial and parallel code

While the code that is implemented on the host is written in C the GPU parallel code that optimizes the performance is written in CUDA C. The execution of a standard CUDA code is carried out in the following steps:

1. Transfer of data from the memory of the CPU to the GPU.
2. Initiation of the kernel code to process data present in the GPU memory
3. Transfer of data from the memory of the GPU to the memory of the CPU after the processing is complete.

3. CUDA AND IMAGE ANALYSIS

Another domain that can greatly benefit from the CUDA programming model is image analysis. The pixel content of any modern high definition digital image can be in the excess of three million and the current image processing software utilities may need multiple computations per pixel. This creates a massive computational load on the processors and could easily lead to inefficient run time and high latency. Such massive processing loads could seriously affect the processing capabilities of the CPUs leading to decrease in productivity. The CUDA programming model can dedicate a single process thread per pixel and can greatly decrease the image processing runtime and increase efficiency. The single thread that is assigned to a single pixel will be responsible for processing its final color output. Considering the fact that the digital images are 2D in nature, CUDA allows process thread wraps and blocks to be two dimensional as well. As a result, a standard process thread block could be in the size of 32X16 and this will allow concurrent running of 512 different process threads at any given point of time. Furthermore, it is possible to add or start as many thread blocks as required in both the dimensions to ensure that the entire image is processed.

A natural inquisitiveness at this stage could be on the size of thread blocks as to why they are not 32X32 instead of the 32X16 size as mentioned above. A simple answer to this question lies in the architecture of the CUDA systems that only accommodates 512 different process threads. So it will not be possible to handle 1024 threads that would come with the 32X32 size. However the above is only true for CUDA versions 1.3. With the CUDA system version 2, it is possible to accommodate 1024 process threads per block, thanks to extensive enhancement of the CUDA 2 hardware architecture. Due to the fact that the standard CUDA systems does not come with the hardware enhancements of version 2, it is always safe to implement 512 process threads per block to ensure efficiency and productive of the parallelization that comes with the CUDA programming model.

Implementation of CUDA paradigm for digital image processing is relatively simple to implement and requires only minimal annotation of the code to invoke the parallelization process. A simple code snippet that is typically executed by image processing algorithms is shown below:

```
for (int i=0; i < height; i++)
 {
for (int j=0; j < width; j++)
   {
result[i*width+j] = ProcessPixel(j,i);
   }
}
```

Annotation of this code for invoking CUDA requires replacement of the for loops in the above code with the thread and block id calculations shown below:

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

During image processing on a CUDA system, a copy of the data of the digital image is made in the Graphics processing unit memory. This is followed by another copy of the image data in another location in the Graphics processing unit memory followed by the invocation of the kernel code. Once the processing is complete and the final image data is ready, it is copied back to the host memory.

In the last decade there has been a lot of interest in exploiting the capabilities of the CUDA programing model for image processing and analysis. Most of them are focused on image edge detection and digital image segmentation that are extremely resource and time intensive. A good example of CUDA implementation

for image analysis and processing was presented in a study where the authors carried out histogram equalization, edge detection, cloud removal and encoding/decoding of DCT on digital images (Yang, *et. al,* 2008). For this purpose the researchers carried out the segmentation of the image data that was sent as the input into process threads followed by the computation of the probability density function. The density function is required for digital image equalization and the same is calculated for input data that comes in the form of a subset. Processed results for every thread is finally collated to form the output data and presented to the programmer. The CUDA approach that was utilized in this study to carry out the image histogram equalization is illustrated in **(Fig, 4)** below.

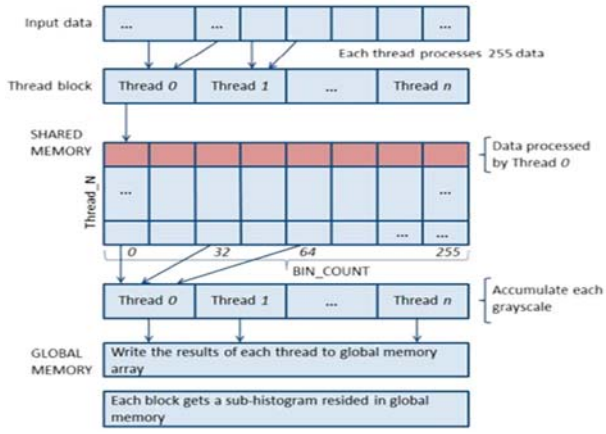

Fig.4. CUDA mediated histogram equalization.

It is interesting to note that majority of the studies that focus on image analysis processes are based on the approach described above when it comes to integrating the same to the CUDA implementation process. The fundamental step is to split the input data into multiple threads as in the study above followed by processing and assembly of individual thread results into the final result. Another study that is based on this philosophy focussed on the efficacy of parallelization in detecting the image contours while others presented unique implementation of edge detection algorithm proposed by Canny through the CUDA approach (Catanzaro, *et. al,* 2009), (Luo, *et. al,* 2008), (Park, *et. al,* 2008). With regards to the Canny's edge detection algorithm using CUDA one study presented a comparative evaluation of the different CPU implementations while the other showed efficacy in the edge detection process brought about by the CUDA mediated parallelization (Luo, *et. al,* 2008), (Park, *et. al,* 2008).

Given the phenomenal increasing in the parallel processing capabilities of the Graphics processing units in the recent years, another domain that is a witnessing a massive surge in CUDA mediated application development is modern medical science. In this regard a study carried out by Boyer et al. demonstrated that white blood cell detection and tracking in microscopic images can be fastened by up to two hundred times through a Graphics processing unit implementing the CUDA programming model (Boyer, *et. al,* 2009). Another interesting approach presented in a different study involved an intuitive integration of the CPU and GPU architecture for the purpose of biological image analysis. The proposed image analysis application can invoke multiple kernel processes and contain operators for data streaming, image convolutions and even histogram processing (Hartley, *et. al,* 2008). Other studies presented interesting implementation of biological image segmentation applications through the use of the CUDA paradigm and CUDA compatible Graphics processing units (Pan, *et. al,* 2008), (Ruiz, *et. al,* 2008). It can be stated that the current CUDA approaches are facilitating the parallel processing of the standard image analysis operations but there is a pertaining need and scope for further development in the domains of biological image processing.

Another important image processing technique is convolution filtering that can be utilized for smoothing of digital images and image edge detection. While convolutions find extensive mention and utilization in the engineering domain, they are also being used in blur filters and edge detection of digital images. A convolution filter that is two-dimensional in nature need the kernel height and width multiplication for every pixel that is sent as an output. When the convolution filters are separable in nature they integrate a one-dimensional filter for the rows and another one for the image columns.

For the purpose of utilizing the CUDA paradigm for implementing convolutions the first step is load an image data block to a memory allocation that is shared. This is followed by the height and width multiplication of a data section that is the size of a convolution filter and addition of the value to the output image data within the memory of the device as shown in **(Fig. 5)** below. As it can be seen in the figure, the execution of the convolution algorithm involves loading of an image pixel block onto an array located in a shared memory allocation.

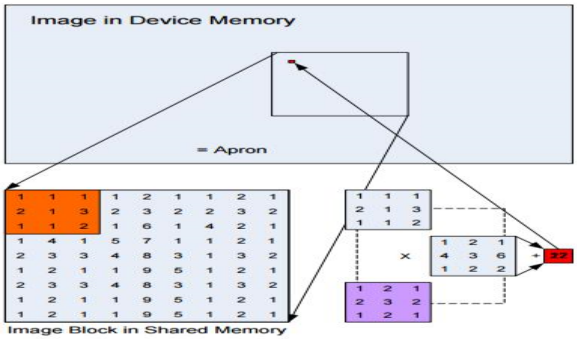

Fig.5. Schema of the convolution algorithm

For the calculation and deduction of an output pixel shown in red, a multiplication of an image section that is sent as an input and shown in orange is carried out within the kernel shown in purple. The output pixel that is retrieved is engraved back onto the main image.

Notes under each table and figure should be used to explain and specify the source of all data shown.

4. CONCLUSION

The 21st century is witnessing a phenomenal rise in the generation of digital data and presenting unique data processing challenges to the scientific community. An innovative and effective approach to deal with the processing load of this massive data surge is parallelization of the computational processes. The strength of the CUDA programming model lies in the possibility of extensive parallelization of the computational processes in the form of threads, wraps and blocks and their concurrent processing across multiple cores. This not just decreases the computational load but also speeds up data processing and optimize performance. While CUDA finds application in multiple domains, exciting avenues are being explored in the area of image processing and analysis. This study presents an exhaustive review of the CUDA programming approach and the the implementation of CUDA for digital image processing and analysis, It is anticipated that the knowledge gained from this study will serve as an excellent know base that would stimulate conceptualization of innovative ideas in future.

REFERENCES:

Abecassis, F., S. Lavernhe, C. Tournier and P. Boucard, (2015). "Performance evaluation of CUDA programming for 5-axis machining multi-scale simulation", Computers in Industry, vol. 71, 1-9.

Boyer, M., D. Tarjan, S. Acton and K. Skadron, (2009). "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors", 2009 IEEE International Symposium on Parallel & Distributed Processing.

Cheng, J. M. Grossman and T. KcKercher, (2014). Professional CUDA C programming, 1st ed. Indianapolis: Wrox,. 2-110.

Catanzaro, B., B. Su, N. Sundaram, Y. Lee, M. Murphy and K. Keutzer, (2009). "Efficient, high-quality image contour detection", 2009 IEEE 12th International Conference on Computer Vision

De Donno, D., A. Esposito, L. Tarricone and L. Catarinucci, (2010). "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook", IEEE Antennas and Propagation Magazine, vol. 52, no. 3, 116-122.

Gu J. and L. Gu, (2006). “Fast DDR Generation Based on GPU,” International Journal of Computer Assisted Radiology and Surgery.

Hartley, T., U. Catalyurek, A. Ruiz, F. Igual, R. Mayo and M. Ujaldon, (2008). "Biomedical image analysis on a cooperative cluster of GPUs and multicores", Proceedings of the 22nd annual international conference on Supercomputing - ICS '08.

Luo and R. Duraiswami, (2008)."Canny edge detection on NVIDIA CUDA", 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops.

Park, S., S. Ponce, J. Huang, Y. Cao and F. Quek, (2008). "Low-cost, high-speed computer vision using NVIDIA's CUDA architecture", 37th IEEE Applied Imagery Pattern Recognition Workshop.

Pan, L., L. Gu and J. Xu, (2008). "Implementation of medical image segmentation in CUDA", 2008 International Conference on Technology and Applications in Biomedicine.

Ruiz, J. K., M. Ujaldon, K. Boyer, J. Saltz and M. Gurcan, (2008). "Pathological image segmentation for neuroblastoma using the GPU", 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro.

Sharp, G., N. Kandasamy, H. Singh and M. Folkert, (2007). "GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration", Physics in Medicine and Biology, vol. 52, no. 19, 5771-5783.

Vetter C. and C. Guetter and C. Xu and R. Westermann, March, (2007). “Nonrigid multi-modal registration on the GPU,” Medical Imaging 2007: Image Processing, SPIE, vol. 65, 12.

Yang, Z., Y. Zhu, Y. Pu, (2008). “Parallel Image Processing Based on CUDA”, 2008 International Conference on Computer Science and Software Engineering, 2008 Dec. 12-14, 198-201.