Performance Analysis of CUDA and OpenCL By Implementation of Cryptographic Algorithms

Manas Mahapatra

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India
May 2015

Performance Analysis of CUDA and OpenCL By Implementation of Cryptographic Algorithms

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Information Security)

by

Manas Mahapatra

(Roll- 710CS2158)

Under the supervision of

Prof. Korra Sathya Babu

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, 769 008, India

May 2015

Declaration by the Student

- The work enclosed in this thesis has been done by me under the supervision of my project guide.
- The work has not been submitted to any other Institute for any degree or diploma.
- I have confirmed to the norms and guidelines given in the Ethical Code of Conduct of National Institute of Technology, Rourkela.
- Whenever I have adopted materials(data, theoretical analysis, figures and text) from other authors, I have given them due credit through citation and by giving their details in the references.

Date: 01 June 2015 **Manas Mahapatra**

Place: Rourkela

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled ***Performance Analysis of CUDA and OpenCL by Implementation of Cryptographic Algorithms*** by ***Manas Mahapatra*** bearing roll number 710CS2158, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering.

Place: NIT Rourkela
Date: 01 June 2015

Prof. Korra Sathya Babu
Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

All the effort I have put forward in carrying out this project would have been incomplete, if not for the kind support of many individuals as well as this institute. I would like to express my deep sense of gratitude to all of them. Foremost, I would like thank my supervisor of this project, Prof. Korra Sathya Babu, Department of Computer Science and Engineering, National Institute of Technology, Rourkela, for his incalculable contribution in the project. He stimulated me to work on the topic and provided valuable information which helped in completing the project through various stages. I would also like to acknowledge his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis.

I am obliged to all the professors of the Department of Computer Science and Engineering, NIT Rourkela for instilling in me the basic knowledge about the field that greatly benefitted me while carrying out the project and achieving the goal. I thank the admin and sta of National Institute of Technology Rourkela for extending their support whenever needed.

I also thank my friends and peers who have extended their help whenever needed. Their contribution has always been significant. Finally, I would like to take this opportunity to thank my parents, who have always been a source of inspiration and motivation for me, and also for the love they have provided in stressful periods, which has been a guiding force for the completion of the thesis.

Manas Mahapatra

Roll-710cs2158

Abstract

This paper presents a Performance Analysis of CUDA and OpenCL. Three different cryptographic algorithms, i.e. DES, MD5, and SHA-1 have been selected as the benchmarks for extensive analysis of the performance gaps between the two. Our results show that, on the average scenario, CUDA performs 27% better than OpenCL while in the best case scenario it takes over OpenCL by 30%.We also infer that CUDA is more stable and completely masks the access latencies to the shared memory due to the contention of 16 read ports. As far as the optimal number of threads per block goes, 256 threads per block is the most performant choice, proving that the CUDA architecture is able to deal with an increased pressure on the register file without problems as CUDA scores 4.5times over OpenCL in terms of stability.

Keywords: Performance Analysis, DES, MD5, SHA-1, CUDA, OpenCL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid development of computing ability on consumer grade hardware, especially in the area of using Graphics Processing Units (GPUs) for general purpose computing using OpenCL, CUDA has rendered today's enthusiast PC at or near the level of the super computers of the late 90s.Parallel computing platforms and programming models like OpenCL and CUDA have the advantage to provide an application a bypass to a graphics processing unit which can be used for non-graphical computing. A graphics processing unit (GPU), is a particular electronic circuit intended to briskly control and adjust memory to speed up the formation of pictures in a frame buffer. Scholarly scientists have researched naturally gathering these projects into application-particular processors running on FPGAs and business FPGA sellers are creating apparatuses to make an interpretation of these to keep running on their FPGA gadgets [11].

As, more and more multi-core processors are taking over sequential ones, increasing parallelism, rather than increasing clock rate, has become the chief appliance for growth of performance [12] . Developers and scientists are really turning out to be progressively intrigued by saddling this power for universally useful registering, an exertion referred to all in all as GPGPU (for General-Purpose computing on the GPU) [13].

Owing to this tremendous performance prospective, GPU programming models have transformed from high-level languages such as HLSL [14],Cg [15],and GLSL [16] to recent programming languages, which has successively increased programmers load and thus enhanced GPUs acceptance. The launch of CUDA

(Compute Unified Device Architecture) by NVIDIA in 2006 has diminished the level of use of the graphical APIs for computational activities, resulting in widespread use of GPU computing [9]. Similarly, a programming framework known as APP (Advanced Parallel Processing) that is known to enable ATIs GPUs in concurrence with the CPUs speeds up a number of requests [17]. These agendas in terms have enabled the programmers to cultivate GPU computing application without much knowledge on graphics.

Since, the method for application development varies from one programming network to another inconvenience arises as the software development and its counterparts have to be built again from the very beginning each time a new platform is launched. This in turn gave rise to an Open Standard known as OpenCL(Open Computing Language) overseen by the Khronos Group which allowed parallel programs to be executed across heterogenous stages giving programming designers versatile and productive access to the force of diverse processing platform.

OpenCL gives a versatile dialect to GPU programming focusing on extremely different parallel processing gadgets. Not at all like a CUDA portion, it possesses a unique feature of being compiled at runtime. Despite what might be expected, this in the nick of time arrange may permit the compiler to produce code which would improve utilization of target GPU.CUDA can be utilized as a part of two distinct ways, extensions furthermore by means of the driver API, which gives low level control over the equipment and through runtime API that gives a C like arrangement of runtimes. CUDA being produced by the same organization that adds to the equipment it executes on is required to perform better coordinating the processing attributes of the GPU. Considering these variables it is of most extreme enthusiasm to contrast OpenCL's execution with that of CUDA in genuine applications. In this paper cryptographic algorithms are applied to investigate the performance of CUDA and OpenCL. Comparison is carried out on NVIDIAs GPU, since OpenCL is still immature and comparison against 5-yr old IBMSDK would be clearly partial by intention. Moreover, on the multicore processor, there is no model with comparable low-level granularity. In this way, the distinction in

execution can be ascribed to the proficiency of relating programming structures

1.1 Motivation

With the advent of technology multi-core processors are dominating the sequential ones. This in a way has shifted the focus on increasing parallelism rather clock-rate. The necessity of using the graphical APIs for computing applications has been eliminated by the release of CUDA by NVIDIA in 2006.Likewise,other programming models like APP came into existence allowing software engineers to cultivate GPU computing applications without ardent knowledge graphical terms. Since, these frameworks had their unique method of applications development, it was quite inconvenient for software developers as they had to rebuilt everytime a new platform was launched. This resulted in the development of OpenCL by the Khronos group allowing parallel programming to be executed across heterogeneous platforms. Nevertheless, this raised the question of performance compromise which is frequently the case with these type of common languages.

1.2 Objective

We emphasize on the performance comparison of CUDA and OpenCL by implementation of cryptographic algorithms as in our view this would be the most relevant comparison. Firstly, since OpenCL is still immature on the Cell Broadband Engine and it would be quite unfair to compare it against the 4-yr old IBM SDK. Secondly, both CUDA and OpenCL are inviting more and more attentiveness from prationers and researchers and in a way share core ideas in terms of memory, platform, execution and programming models.

1.3 Thesis Contribution

1. A method has been proposed for performance analysis of CUDA and OpenCL.

2. Three different cryptographic algorithms, i.e. DES, MD5, and SHA-1 have been selected as the benchmarks for extensive analysis of the performance

gaps between the two.

3. In order to analyze the performance of CUDA and OpenCL, a normalized Performance Metric called as Performance Factor(PF) has been defined.

   - $$PF_{1Avg.} = \frac{Average\,Throughput_{CUDA}}{Average\,Throughput_{OpenCL}} \tag{1.1}$$
   - $$PF_{1Max.} = \frac{Maximum\,Throughput_{CUDA}}{Maximum\,Throughput_{OpenCL}} \tag{1.2}$$
   - $$PF_{2} = \frac{Percentage\,Increase\,in\,Throughput^{-1}_{CUDA}}{Percentage\,Increase\,in\,Throughput^{-1}_{OpenCL}} \tag{1.3}$$

     Where, PF_1AVG ,PF_1MAX ,and PF_2 compare average performance, peak performance and relative stability respectively.

1.4 Thesis Organization

- **Chapter-2** In chapter 2, we have given a brief idea of GPU architecture and an overview of OpenCL and CUDA and made a comparison between the two. The Cryptographic algorithms which are implemented for performance analysis are also discussed vividly.

- **Chapter-3**, In this chapter the Methodology and Experimental Setup adopted for the analysis have been discussed.The parameters used in Performance Metric have also been pondered through

- **Chapter-4**,This chapter deals with the implementation and analysis of the pre-described algorithms in OpenCL and CUDA.Peak Performance, Average Performance and Stability are taken into account for grading.

- **Chapter-5**, In this chapter,based on the results obtained we arrive at certain conclusion.

Chapter 2

Literature Survey

2.1 GPU Architecture

GPUs a.k.a Graphical Processing Units are processors determined to diminish the pressure on the CPU, when working on video graphics. Over the years, GPUs have emerged as a vital component of computing platforms. GPUs can be perceived as accelerators as well as co-processors. They do not essentially preclude the demand for a CPU. Though CPUs primary job is to execute serial applications as fast as possible but lately CPUs have evolved as multicore with an ability to achieve multithreaded parallelism, the only backdrop being they are still optimised for serial execution. On the part, GPUs are rather dedicated for the act of densely threaded parallelism [2].

Recently, GPUs have evolved as a more general pur-pose computing element from being just video and graphic accelerator referred to as General Purpose Graphics Processing Units (GPGPUs) [8]. This course initiated with shader languages has lately transformed into an entire series of development tools to ease general purpose GPU computing whose prime vendors being nVidia, AMD (formerly ATI) and Intel. GPUs being highly parallel, multithreaded programmable devices with an ability to render real-time graphical applications have thousands of cores with tremendous power capable of high precision floating point arithmetic the very necessity of real-time video processing [18]. Graphics cards possessing a very high memory bandwidth allow huge amounts of data to be transferred in a single flow coupled with a large number of on chip registers which hold several

variable values while computation takes place [13].

GPU have an exponentially higher number of floating point operations per second(FLOPS) as compared to a high end CPU which is crystal clear from the fact that computation speed of a standard GPU is in few hundred gigaflops while a high end CPU possesses few tens of gigaflops. GPUs proficiency in performing compute intensive highly parallel tasks can attributed as the very reason for such high computation speed .The GPUs are extensively being used for parallel applications i.e. applications where the problem is divided into number of parallel tasks especially when the arithmetic operations exceed memory operations. Dataflow between processors is reduced since the same program is completed on dierent processing elements on dierent data sets. As a result, memory latency remains hidden under the heavy calculations that take place inside the processors . Applications where usage of large data sets is high priority benefit from employing parallel programming model [1].

2.1.1 GPU Processing Elements

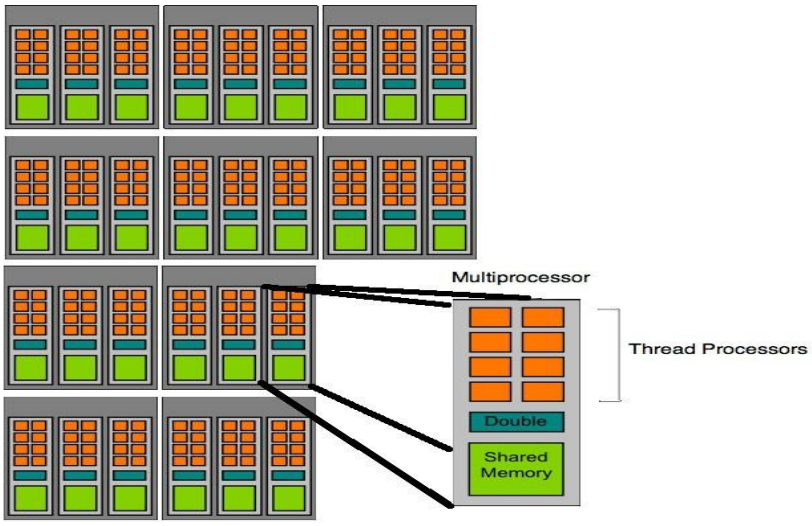

Figure 2.1: Diagram of multiprocessors in GPU [1]

The GPUs bear a few Streaming Processing Clusters (SPC). All SPCs comprise numerous streaming multiprocessors (SM), each one of which contains streaming processors (otherwise called cores) that impart admission to local memory. Each

core contains an intertwined multiply-adder for single exactness number-crunching [2].

2.1.2 GPU Memory Organization

Every multiprocessor has a 16KB region of shared memory space with short access times. The reason for shared memory is to go about as a methods for quick correspondence between threads. Be that as it may, because of its speed, it can likewise be utilized as a software engineer controlled memory cache.

GPUs have DRAM (Dynamic Random Access Memory).DRAM are available at moderately 150x inactivity in correlation to shared memory. This memory is consistently divided into four regions: local memory, global memory, constant memory and texture memory. Global memory is steady between GPU calls as it is quite convenient to all threads.It dwells o chip from the multiprocessors, bringing about 100x access time contrasted with shared memory. Local memory, particular to individual threads can likewise be utilized as a substitute in case the compiler is inadequate to force in sought information into the gadget's registers. Texture memory being read-only is availed with texture cache for texture manipulation. It is beneficial than global memory since the memory peruses don't oblige an access pattern to get better execution and computations are done outside the kernel. Constant memory is additionally a read-only part which likewise has a small cache of 8K.

Toward the end, host memory (framework's primary memory) is open at a slant and relatively slower to the GPU. Host memory space is convenient just to the GPU when duplicated over the PCI-Express (Peripheral Component Interconnect Express) bus to the GPU's device memory.

2.2 OpenCL Overview

OpenCL is an open standard focused to give programming designers a standard structure for simple access to different heterogeneous preparing stages that include exceptionally parallel GPUs, CPUs and different sorts of processors. The

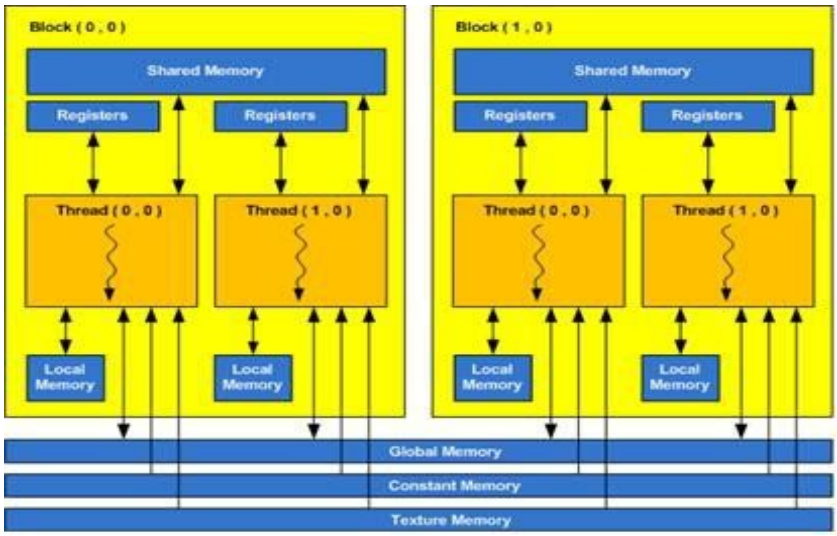

Figure 2.2: GPU Memory Hierarchy [2]

OpenCL standard specifies a programming standard taking into account C and an arrangement of API. The details about the OpenCL structure can be found in the OpenCL specifi- cation [19]. The OpenCL structure can be best depicted by the four models

1. Platform Model
2. Execution Model
3. Memory Model
4. Programming Model

2.2.1 Platform Model

The OpenCL platform model as given in Figure 2.3 comprises of a host which is typically a CPU associated with one or more OpenCL Compute Devices which can be a CPU or a GPU. A Compute Device is a mix of Compute Units, which are further isolated into Processing Elements, which arries out the real processing.

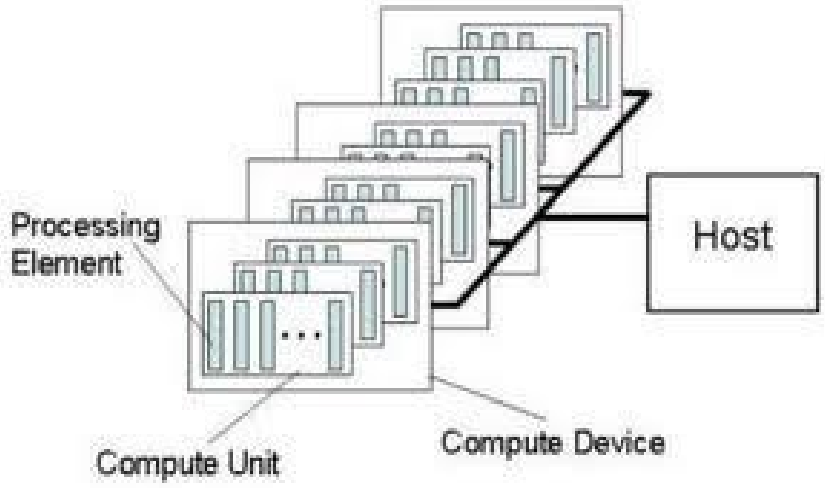

Figure 2.3: OpenCL Platform Model [3]

2.2.2 Execution Model

The execution of an OpenCL system can be isolated in two sections: the device code which runs on one or more Compute Devices and the host code which runs on the host gadget. Kernels and memory objects are overseen by the host part under a connection through command queue

1. Context:

   The context constitutes of every last one of pieces important to utilize a gadget for processing reason. By utilizing the OpenCL API, the host part of the code makes a context object and subsequently different objects under it, i.e. program object, kernel object, command queues object, and memory objects.

2. Kernel:

   The calculation that is executed on the processing elements is spoken to by kernel. A case to illuminate the kernel idea is described. Accepting there is a number cluster, i.e. an integer array of size 100 and the objective is to add every integer by a constant. Kernel for this issue would just speak to represent addition of one integer number by the constant , instantiating the kernel 100 times to tackle the complete issue. In any case, out of thought

for processor use and memory access, it is conceivable to add two integers in the same kernel. In the event that that is the situation, the kernel would be instantiated fifty times to tackle the complete issue.

3. Work Items and Work Groups:

   Kernel execution on a gadget is characterized by a list space, called NDRange. A NDRange is a N-dimensional list space, where N can fluctuate from one to three. The kernel instance is known as a work-item. All the work-items concentrate on the same piece of code. In any case, they as a rule take a shot at distinctive information and there may be dissimilarity in their execution way through the code. Every work-item is allocated a global ID which is novel all through the indexed space.

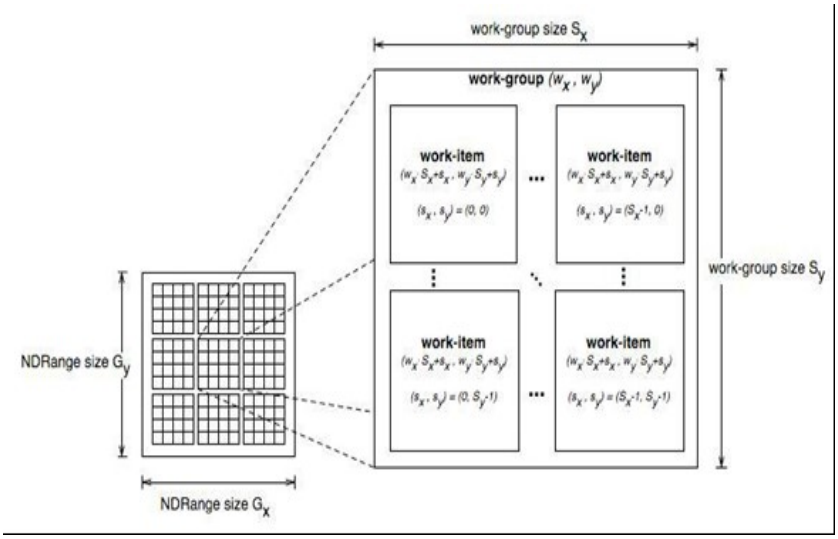

Figure 2.4: OpenCL Execution Model [4]

The equivalent number of work-items are assembled together to frame a work-group. All the work-groups have same measurements. The work-item inside a work-group has a nearby ID that is exceptional over the work-group, furthermore has entry to shared local memory. It is important to note here that with fitting gadget bolster, the aggregate number of work-items may be much more prominent than the quantity of handling components show in a gadget.

4. System and Memory Object:

   The program object constitutes of the source code and the binary implementation of the kernels. The binary implementation can be generated from the source code during application execution or a pre-compiled binary can be loaded to create the program object. A program object is a library for kernels because one program object may contain multiple kernels. Decides of which kernel to execute during execution is done by application during runtime.

   The host creates memory objects, and through the OpenCL API, memory is allocated on the device for the memory objects. The memory model is described in detail in the next section.

5. Command Queue:

   The command queue is connected with every gadget in the connection, and memory exchange and kernel execution are facilitated utilizing it. There are three sorts of commands which can be issued. Memory orders are generally used to exchange memory between the host and the gadget. Kernel commands are utilized to begin the execution of kernels on the gadget. Synchronization commands are utilized to control the execution requests.

   When the commands have been booked on the queue, there are two conceivable methods of execution. First one defined as in-order where current command can start execution only if previous command has finished its part. The other alternative being out-of-order execution. Here, commands don't sit tight for beforehand lined commands to complete execution.

2.2.3 Memory Model

The memory model utilized inside a Compute Device is demonstrated in Figure 2.2.3. The execution model discussed in 2.2.2 is plotted in this model. The mapping of work-group happens onto a Compute Unit, while a work-item executes on a PE (Processing Element). Work-items executing a piece have admittance to

different locales of memory. Global memory gives authorization of read/write access to all work-items of each work-group.

Moreover, work-items belonging to the same work-group have access to the local memory. Contingent upon the gadget capability, local memory can be mapped onto the dedicated memory locales of the gadget or onto the segments of the global memory.

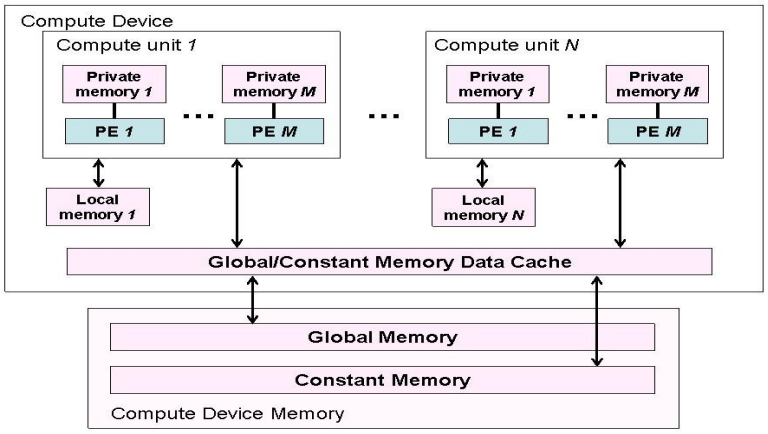

Figure 2.5: OpenCL Memory Model [3]

2.2.4 Programming Model

Under the OpenCL programming model, calculation can be performed in task parallel, data parallel, or a crossover of these two models. The real center of the OpenCL programming model is the data parallel model, where every work-item chips away at an information thing actualizing SIMD.

The task parallel model can be acknowledged by enqueing various kernel execution, where one and only work-item is made for every part. Despite the fact that a couple GPUs give backing to this model, this is profoundly wasteful model for the GPUs.

A hybrid model is conceivable where various bits each with numerous work-items are enqueued for execution in the meantime.

1. Execution Flow in an OpenCL Application:

The OpenCL application stream is indicated in Figure 2.2.4. The stream is separated into two areas. A context is made by platform layer taking into account accessible platforms, and the runtime layer makes all other vital items expected to execute the piece.

2. Platform Layer:

   In an OpenCL application, at first an inquiry is made for accessible OpenCL plat-forms. Once the accessible platform rundown is gathered, the application picks the one with the compatible device type and a context is made. The conceivable gadget sorts allowable in the OpenCL specification are cl device type gpu, cl device type cpu, and cl device type accelerator. The craved number of gadgets from the accessible gadgets is included by the setting. The gadgets are made selective to the setting once added to a context until they are expressly discharged from the context.

3. Runtime Layer:

   The depiction of errands thought to be a piece of the run-time layer is given beneath. Host and the gadgets impart one another utilizing the commands. A command queue is made for every gadget under the connection to issue commands. A discretionary OpenCL object can be made, at whatever point an order is issued. These event objects can be utilized for explicit synchronization and permit the application to check for the finishing of the command. To distribute memory on the gadgets, memory objects are made. The application sets the authorization to read/write with these memory objects from the host when they are made. By either stacking the source code or by the twofold usage of one or more kernels, projects objects are made. The binary representation can be middle of the road representation or the gadget specific executable. The program objects are then fabricated to create the gadget specific executable. The OpenCL usage chooses of the move to be made in the manufacture stage depending on whether source code, transitional representation, or an executable was utilized to make the

system object.

The arrangement of the output is not under the OpenCL specification, and the OpenCL execution chooses an organization of accommodation. The kernel object is made once the executable is assembled in the program object. One of the capacities actualized in the project item is spoken to by the kernel object.

The information is exchanged to the gadget memory by issuing memory duplicate commands against the associated memory objects before executing the kernel. The memory exchange can either be blocking where once the memory exchange is finished, the control is come back to the application or non-blocking where control is returned after the memory exchange is booked. The occasions are utilized for synchronization for a non-blocking exchange. The estimations of the kernel arguments are situated once the information is exchanged and the kernel through the command queue is planned for execution. The output memory is exchanged to the host from the gadget once the portion execution is finished. We can have an iterative methodology where the same portion is planned to run once more. New data information can be exchanged to the gadget, and after kernel execution new output information can be exchanged back to the host.

2.3 CUDA Overview

Compute Unified Device Architecture, or CUDA, is NVIDIAs programming model and parallel configuring stage. It is a full processing stage with an equipment structural planning specification, which is upheld by expanded variants of programming dialects. The CUDA equipment is in view of the innovation of GPU. The GPU, or graphical processing unit, was authored by NVIDIA in 2000 [8]. During this period, VGA controllers were progressing to bolster quickening of 2D- and 3D-illustrations, and the GPU presented an incorporated preparing unit that upheld that of a conventional top of the line workstation representation pipeline, subsequently there was a requirement for a term. From that point forward, GPUs

have relentlessly ended up more broad, supplanting function rationale with programmable usefulness [18]. The first employments of GPUs for universally useful registering (GPGPU) were acquired by misusing design programming APIs , for example, the open source OpenGL and Microsofts DirectX libraries. This was made conceivable by the distinct behavior of the APIs. The disservice was that the client expected to have private information of the APIs and the capacity to express projects regarding representation.

To tackle the issues related to GPGPU programming, NVIDIA introduced the unified gadget building design, discharged CUDA C, a variant of standard C with the expansions to bolster GPU programming. The first proficient gadget of CUDA, speaking to CUDA capability v1.0 was the G80 structural planning, which was first discharged in 2006. From that point forward new CUDA-based architectures have included highlights bringing about upgrades of the ability specification of CUDA, trailed by backing in CUDA C.

2.3.1 CUDA Programming Model

The programming model given by CUDA has permitted engineers to utilize the force of the versatile parallel processors without breaking a sweat, empowering them to accomplish velocity ups of a few times on an assortment of uses. Since the arrival of CUDA by NVIDIA in 2007, a great deal of versatile parallel projects were quickly developed for an expansive scope of uses, including sorting, network solvers, looking, material science models and computational science.

CUDA gives some effortlessly comprehended deliberations that permit the programmer to concentrate on the efficiency of the algorithm and create versatile parallel applications by outflow of parallelism expressly. It gives three key abstractions which is a chain of importance of thread groups, shared memory, and synchronization which give an unmistakable structure to the routine C for one string of the pecking order. The reflections direct the developer to break the issue into coarse sub-issues that can be unraveled freely in parallel, and afterward into subsequent pieces that can be understood in parallel helpfully. The programming model scales to substantial quantities of processor centers straightforwardly: an

accumulated CUDA project can execute on any number of processors, and physical processor tally needs to be known by run time environment [20] [19].

As was clarified some time recently, CUDA can likewise bolster heterogeneous computation. The serial piece of the applications is run on the CPU, and parallel bits are loaded to the GPU. The CPU and GPU are dealt with as discrete gadgets which have their own memory spaces. This configuration likewise permits synchronous and covered calculation on both the CPU and GPU without controversy for memory assets. The irreplaceable piece of the code for CUDA is the kernel program. which works on the whole stream of information. The setting of a CUDA piece is basically a C code for one thread of the pecking order, however execution is in parallel over an arrangement of parallel threads. These strings are masterminded into a progression of a matrix of thread blocks. A network is a situated of thread blocks that can be autonomously transformed on the gadget by planning blocks for execution on the MP and accordingly, they may execute in parallel and threads of a block can just get to the shared memory. The execution of thread block happens as littler gatherings of threads known as "warps". Thus, individual threads that form a warp begin together at the same system address yet they are allowed to execute and branch autonomously.

CUDA backings thread blocks can contains up to 512 threads. The thread blocks may have one, two, or three measurements, got to through .x, .y, and .z fields. Parallelism is expressly dictated by indicating the measurements of a lattice and its thread blocks while propelling a kernel. Every kernel dispatch makes a framework of obstructs that allocates one thread to every component of the vectors and conveyance of the threads over the pieces happens. Every thread registers a component list from its thread and block IDs, and the fancied estimation on the comparing vector components is performed. The representation of CUDA programming model as given in [4] is spoken to in Figure 2.7.

CUDA code is by and large straightforward and direct to compose than composing parallel code for vector operations. In any case, while creating CUDA code, it is vital to comprehend the routes in which the CUDA model is limited,

to a great extent for the reasons of productivity. The summon of kernel in CUDA

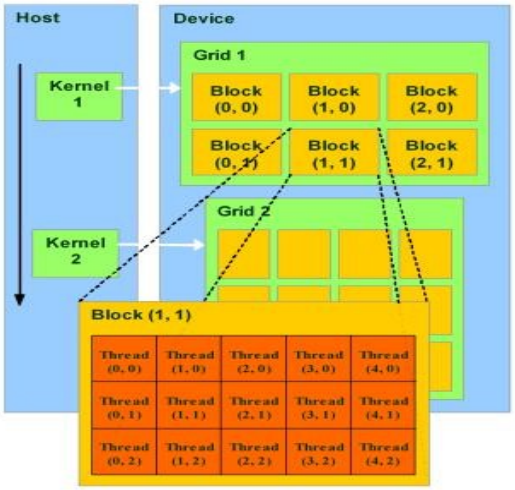

Figure 2.6: Representation of CUDA programming Model [4]

is asynchronous, so the driver will return control to the application when it has propelled the kernel. At the same time, for case, CUDA capacities which perform memory duplicates are synchronous, and they certainly sit tight for all portions to finish.

CUDA code is by and large basic and clear to compose than composing parallel code for vector operations. Anyway, while creating CUDA programs, it is important to comprehend the routes in which the CUDA model is limited, generally for the reasons of effectiveness. The summon of portion in CUDA is asynchronous, so the driver will return control to the application when it has propelled the kernel. At the same time, for example, CUDA capacities which perform memory duplicates are synchronous, and they certainly sit tight for all kernels to finish.

Amid the thread execution, individual threads have admittance to information that settle in diverse memory spaces as given by Figure 2.8. Every thread has admittance to a local memory. Every thread block has an imparted memory to which all threads of the block have entry. Besides, all threads of different blocks can get to same global memory. The texture and constant memory spaces are the two other read-only memory spaces open by all threads: as given in Figure 2.8.

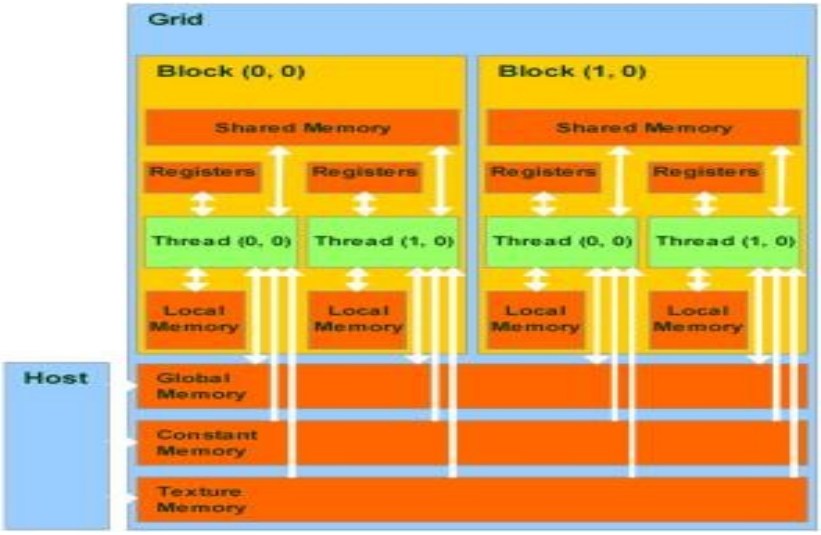

Figure 2.7: Representation of CUDA Threads Blocks mapped on CUDA Memory [4]

2.4 Similarities of CUDA and OpenCL

OpenCL and CUDA share a range of common ideas. They have similar memory, platform, execution and programming models [9] [10]. Table 2.1. describes the necessary details.

Table 2.1: A Comparison of General Terms [9] [10]

CUDA terminology	**OpenCL terminology**
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Thread	Work-item
Thread-block	Work-group

2.5 DES Overview

There are two fundamental sorts of cryptography being used today - symmetric cryptography and asymmetric cryptography. Symmetric key cryptography is the most seasoned sort, though asymmetry cryptography is just being utilized openly since the late 1970's. Asymmetric cryptography was a real turning point in the

quest for an immaculate encryption plan. Secret key cryptography does a reversal to at any rate Egyptian times and is of concern here. It includes the utilization of stand-out key which is utilized for both encryption and decryption (henceforth the utilization of the term symmetric). It is essential for security purposes that the secret key never be uncovered.

To finish encryption, most secret key calculations utilize two principle systems substitution and permutation. Substitution is essentially a matching replacement while permutation is a regrouping of the bit positions for each of the inputs. These methods are used various times as a part of emphases called rounds. Decoding becomes computationally infeasible without the secret key as non-linearity is additionally brought into the encryption. This is attained to with the utilization of S-boxes. One of the primary issues with secret key cryptography is key appropriation. For this manifestation of cryptography to work, both sides must have a duplicate of the mystery key. This would need to be imparted over some safe channel which, shockingly, is not that simple to accomplish [21].

2.5.1 Inner Workings of DES

DES (and a large portion of the other major symmetric figures) is taking into account a figure known as the Feistel block cipher. This was a piece figure grew by the IBM cryptography specialist Horst Feistel in the mid 70's. It comprises of various rounds where each round contains bit-rearranging, non-direct substitutions (S-boxes) and selective OR operations. Most symmetric encryption plots today are taking into account this structure (known as a feistel network) [22].

Similarly as with most encryption plans, DES expects two inputs - the plaintext which is to be encrypted and the secret key. The way in which the plaintext is acknowledged, and the key course of action utilized for encryption and decryption, both focus the kind of figure it is. DES is accordingly a symmetric, 64 bit block cipher as it uses the same key for both encryption and decryption and just works on 64 bit keys of information at a time (be they plaintext or ciphertext). The key size utilized is 56 bits, however a 64 bit (or eight-byte) key is the actual input. The minimum noteworthy bit of every byte is either utilized for equality (odd for

DES) or set subjectively and does not build the security at all. All blocks are numbered from left to right which makes the eight bit of every byte the equality bit [21].

When a plaintext message is gotten to be encrypted, it is masterminded into 64 bit pieces needed for data. On the off chance that the quantity of bits in the message is not equally distinguishable by 64, then the last block will be padded. Various permutations and substitutions are incorporated all through so as to build the trouble of performing a cryptanalysis on the figure. Notwithstanding, it is by and large acknowledged that the beginning and last changes offer next to zero commitment to the security of DES and truth be told some product implementations preclude them (albeit entirely talking these are not DES as they don't hold fast to this standard)

2.5.2 Overall structure

The succession of occasions have been demonstrated that happen amid an encryption operation in Figure 2.9.A permutation is performed by DES on the whole 64 bit piece of information. It is then parted into two 32 bit sub-blocks, known as Li and Ri which are subsequently forwarded to 16 rounds (see figure 2.3), where the subscript i in Li and Ri demonstrates the present round. The rounds are indistinguishable and the impacts of expanding their number is twofold - the algorithm fleeting effectiveness is diminished and its security is expanded. For DES 16 is piked so that the disposal of any connection between the ciphertext and either the plaintext or key is ensured. Toward the end of the 16th round the pre-output is obtained by swapping the 32 bit Li and Ri .Clearly the final permutation gives us the desired 64 bit ciphertext. As deduced from the figure the three basic phrases are as:

1. Initial Permutation (IP - characterized in table 2.1) where the bits are rearranged in order to frame the "permuted data".

2. This is followed by 16 iterations of permutation and substitution. The Last iteration gives a 64 bit output which is a function of plain text and Key.

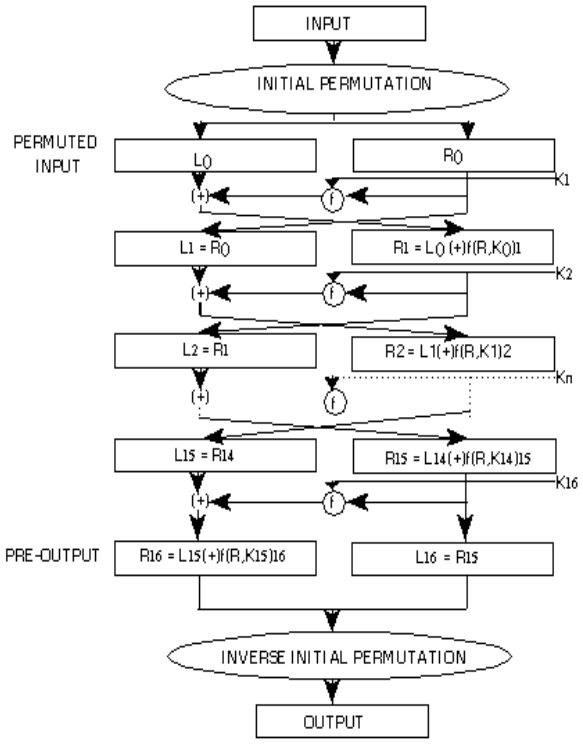

Figure 2.8: Flow Diagram of DES algorithm for encrypting data [5].

Then the right and left halves are exchanged to produce the pre- Output.

3. As, the pre-output is gone through a stage which is essentially the converse of the introductory change IP. The yield of IP^{-1} is the 64-bit cipher.

2.6 HASH Algorithm Overview

A cryptographic hash function has the very advantage of being practically impossible to recreate the input data from its hash value. It is frequently referred to as "the pillars of modern cryptography". The input data is referred as the message, while the hash value the message digest. The idyllic cryptographic hash function has the following properties:

1. The hash value for any message can be easily computed.

2. It is infeasible to generate a message from its hash.

3. It is infeasible to modify a message without changing the hash.

4. It is infeasible to find two different messages with the same hash.

MD5 measures data integrity by the assistance of 128 bit message. Professor Ronald L. Rivest of MIT is the father of this algorithm [23]. The calculation being best suited for 32 bit and 16 bit machines can be stretched out to 64 bit machines. MD5 is a bit slower than MD4 algorithm as MD5 contains four rounds as compared to three by MD4.MD5 being one way hash function arrangements with security highlights. As the dependency on web increases it has become a necessity to ensure that a legitimate record has been download from distributed (P2P) servers/system. The SHA Algorithm is a cryptography hash function which is better utilized as a part of data integrity and digital certificate. SHA is a unique mark that demonstrates its development by N.I.S.T. as a U.S. Federal Information Processing Standard (FIPS) [7].

2.6.1 MD5 Overview

1. Pad up bits and Append Length:

   The message is padded up with zeroes and ones so that the final bit length is equal to 448 mod 512.It is also ensured that the last bit length of the message is 512N for an integer N.

2. Divide the input into 512-bit blocks:

   The message obtained from Step 1 is divided into N progressive 512-bit blocks m1, m2.....mn.

3. Initialize Chaining variable:

   Chaining variables (A,B,C,D) each being of 32 bit size are initialized

   A = 01 17 2d 43

   B = 89 AB CD EF

   C = FE DC BA 98

   D = 76 54 32 10

4. Process blocks:

   A, B, C and D are combined with the input words, utilizing the functions W, X, Y and Z. 16 fundamental operations are iterated via 4 rounds. By utilizing The Message word M[i] and constant K[i] the Processing block P is connected to the four supports (A, B, C and D). Q,W,E,R are the four sort of IRF(Info Related Functions) that apply the sensible administrators $\vee, !$ taking three 32-bit words as input and producing same bits of output i.e. 32-bit word.

   Q (A, S, D) = AS v not (A) F

   W (A, S, D) = AS v S not (F)

   E (A, S, D) = A xor S xor F

   R (A, S, D) = S xor (A v not (F))

   The functions A, S and D = P, as they do work in "bitwise parallel" to deliver the solid yield from the bits of A, S and D.

5. Hashed Output:

   4 rounds are performed in Message Digest 5 (MD5) .

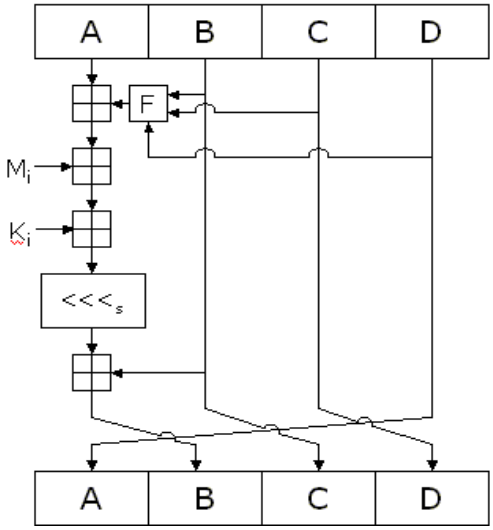

Figure 2.9: One MD5 iteration [6].

2.6.2 SHA-1 Overview

1. Pad up bits:

   Padding is added at the ttermination of the message length so that it becomes a multiple of 512.

2. Append length:

   The appending length is computed in this Step

3. Divide the Input into 512-blocks:

   In this step the input is divided into 512 bit blocks.

4. Initialize chaining variables:

   Chaining variables are initialized here.5 chaining variables of 32 bit each give a total of 160.

5. Process Blocks:

   - The chaining variables are copied
   - The 512 blocks are divided into 16 sub blocks
   - 4 rounds are processed of 20 stages each [6].

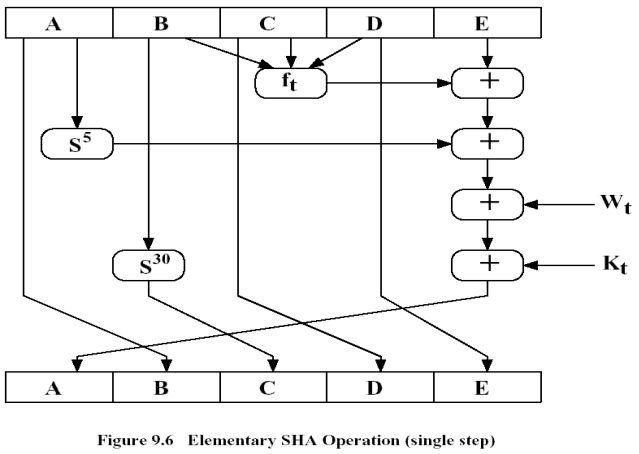

Figure 9.6 Elementary SHA Operation (single step)

Figure 2.10: One SHA-1 iteration [7].

2.6.3 Parameters Used for MD5 and SHA-1 Algorithm

Parameters for MD5

Default Parameters

a = b + ((a + Process P (b, c, d) + M[i] + t[k]) <<< s)

Where:

1. Process P denotes a non-linear operation.
2. a, b, c, d are Chaining variables.
3. M[i] denotes For M[q x 16 + i], which is the ith 32-bit word in the qth 512-bit block of the message.
4. t[k] denotes a constant.
5. <<< s denotes circular-left shift by s bits [7].

Actual Parameters:

Block Size: 128 bits

Key Length: 64 bits, 128 bits, 256 bits , 512 bits

Cryptanalysis: Strong Resistance against Digital Certificate

Steps: 16

Rounds: 4

Parameters for SHA-1.

Default Parameters

a=abcde(e+process $p_s5(a) + W[t] + k[t])$,a,s30(b), c, d

Where:

1. Process p denotes status of logical operations st =<<<
2. a, b, c, d, e denote chaining variables
3. W[t] denotes other 32 bits bytes derived
4. K[t] denotes 5 additives constants as defined in [35] [36].

Actual Parameters.

Block Size: 160 bits

Key Length: 128 bits

Cryptanalysis: Strong Resistance against Digital Certificate.

Steps: 20

Rounds: 4

2.6.4 Comparison between MD5 and SHA

Table 2.2: Comparison between MD5 and SHA

Keys For Comparison	**MD5**	**SHA-1**
Security	Less Secure than SHA	High Secure than MD5
Message Digest Length	128 Bits	160 Bits
Attacks required to find out original Message	2^{128} bit operations required to break	2^{160} bit operations required to break
Attacks to try and find two messages producing the same MD	2^{64} bit operations required to break	2^{80} bit operations required to break
Speed	Faster only 64 iterations	Slower than MD5 Required 80 iterations
Successful attacks so far	Attacks reported to some extents	No such attack report yet

Chapter 3

Methodology and Experimental Setup

In this section, the methodologies adopted in this paper are explained along with the used benchmarks and experimental test beds.

3.1 Performance Factor

In order to examine the performance of OpenCL and CUDA, a normalized Performance Metric titled Performance Factor(PF) has been defined.

- $$PF_{1Avg.} = \frac{Average\,Throughput_{CUDA}}{Average\,Throughput_{OpenCL}} \tag{3.1}$$
- $$PF_{1Max.} = \frac{Maximum\,Throughput_{CUDA}}{Maximum\,Throughput_{OpenCL}} \tag{3.2}$$
- $$PF_{2} = \frac{Percentage\,Increase\,in\,Throughput^{-1}_{CUDA}}{Percentage\,Increase\,in\,Throughput^{-1}_{OpenCL}} \tag{3.3}$$

Where, $PF_{1Avg.}$,$PF_{1Max.}$,and PF_2 compare average performance, peak performance and relative stability respectively.

If PF<1, then performance of CUDA is worse than its counterpart; otherwise, CUDA gives better or same performance. In an instinctual way, if $|PR-1| < 0.1$, it is assumed that CUDA and OpenCL have same performance.

3.2 Selected Benchmarks

Table 3.1: Selected Benchmarks

CLASS	PERFORMANCE,METRIC	DESCRIPTION
Cryptography	Mkeys/sec	DES
Cryptography	Mhashes/sec	MD5
Cryptography	Mhashes/sec	SHA-1

The Benchmarks selected include algorithms frequently used in cryptographic encryptions. Since, these algorithms include complex mathematical calculations GPU performance analysis becomes feasible.

3.3 Experimental Testbeds

The measurements and results are carried out on real hardware on Microsoft Visual Studio platform. Table 3.2,3.3,3.4 denote the hardware environment, software environment and GPU configuration respectively.

Table 3.2: Hardware Environment

Operating Sysyem	Windows
Processor	Intel(R)Core(TM)i5-241@2.30GHz
Installed Memory(RAM)	4.00GB(3.90 usable)
System Type	64-bit Operating System
Graphics Card	NVIDIA GeForce 525M Version 340.62

Table 3.3: Software Environment

Platform	Microsoft Visual Studio 12.0
CUDA version	CUDA 6.5
OpenCL version	OpenCL 2.0
Language	C

Table 3.4: Specifiations of GPU GT525M

Architecture	Fermi
Core	96
Processor Clock Tester(MHz)	1200 MHz
Memory Clock	900 MHz
Memory Interface	DDR3
Memory Interface Width	128-bit
Memory Bandwidth(GB/sec)	28.8

Chapter 4

Implementation and Results

4.1 Implementation

Implementation strategies are adopted in order to fully exploit the computational power of GPU and to generate as many Mkeys/sec or Mhashes /sec as possible.

4.2 Strategy

Crpytanalysis is performed on DES, MD5 and SHA-1 by Brute force attack.Each thread operates on the same pice of code but with a different key value.

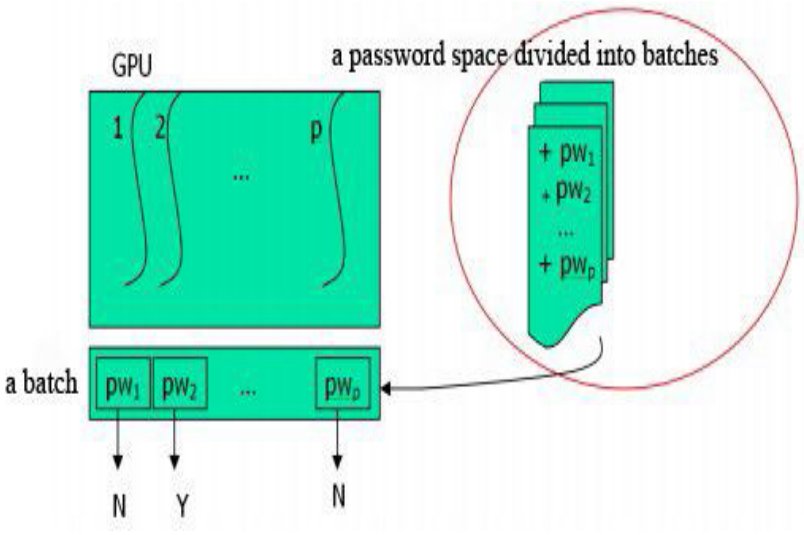

Figure 4.1: Checking passwords in parallel [8]

Since the entire key space cannot be checked at one go as the maximum number of threads per block can only be 1024 so the key space is divided into subsequent blocks.

4.2.1 DES(Data Encryption Standard)

The Data Encryption Standard is one of the most popular encryption algorithms, standardized by NIST in 1977 and subsequently maintained as a FIPS security primitive up to 2005.Table 4.1 gives insight into DES.

Table 4.1: DES General Detail

Designers	IBM
First Published	1977
Derived From	Lucifer
Successors	Triple DES,G-DES,DES-X,LOKI8
Key sizes	56 bits(+8 parity bits)
Block sizes	64 bits
Structure	Balanced Feistel network
Rounds	16

Brute Force Cryptanalysis is carried out on DES with given plaintext and given ciphertext.Keys are varied through the key space and the generated ciphertext is checked with the given one.Keys generate per second by varying threads per block are noted down for efficient comparison.

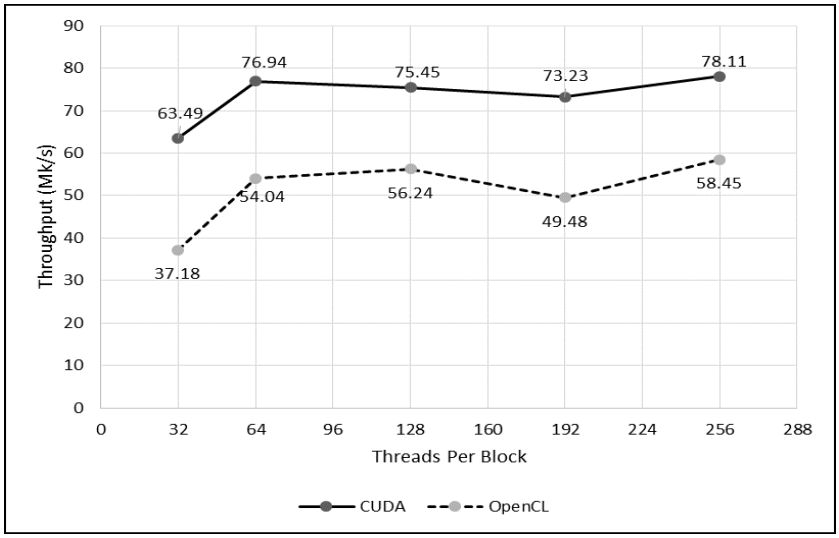

Figure 4.2: Depicts Mkeys/sec generated by varying threads per block from 32 to 256 during implementation of DES in CUDA and OpenCL

4.2.2 Message Digest Algorithm

It is a widely used cryptographic hash function producing 128bit(16 byte) hash value. Typically expressed in text format as a 32 digit hexadecimal number, it is used to verify data integrity. Table 4.2 depicts general description of MD5. Brute

Table 4.2: MD5 General Detail

Designers	Ronald Rivest
First Published	April 1992
Series	MD2,MD4,MD5,MD6
Digest Size	128
Structure	Merkel-Damgard Construction
Rounds	4

Force Cryptanalysis is carried out on MD5 with given MD5 hash.Each thread processes a password and checks the generated MD5 hash against the given hash for a match.Hashes generate per second by varying threads per block in CUDA and OpenCL are noted down for comparison.

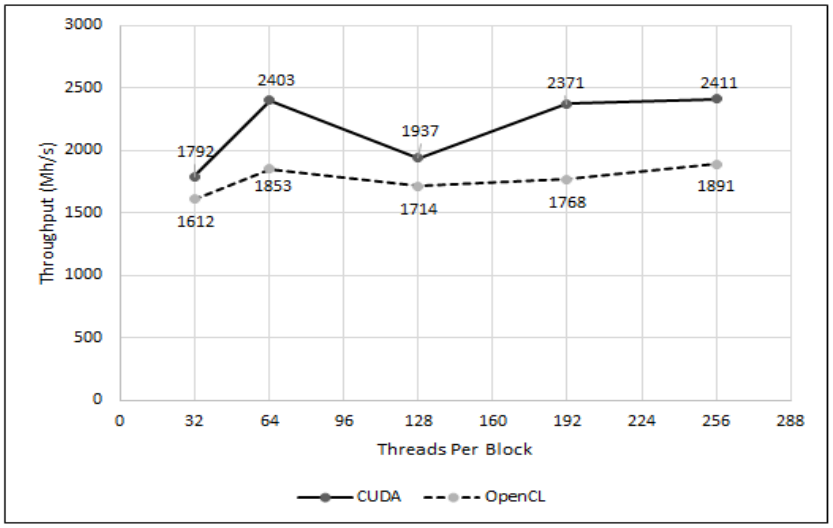

Figure 4.3: Depicts Mhashes/sec generated by varying threads per block from 32 to 256 during implementation of MD5 in CUDA and OpenCL

4.2.3 SHA-1 (Secure Hash Algorithm)

Secure Hash Algorithm is a family of cryptographic hash function published by (NIST) as a US Federal Processing Standard(FIPS) including SHA-0,SHA-1 and

SHA-2.SHA-1 hash function is computed with 32-bit word. Table VIII depicts general description of SHA-1. Brute Force Cryptanalysis is carried out on SHA-1

Table 4.3: SHA-1 General Detail

Designers	National Security Agency
First Published	1995
Series	(SHA-0)SHA-1SHA-2,SHA-3
Digest Size	160
Structure	Merkel-Damgard Construction
Rounds	80

by comparing the generated hash with the given SHA-1 hash. The graph below denotes the results obtained.

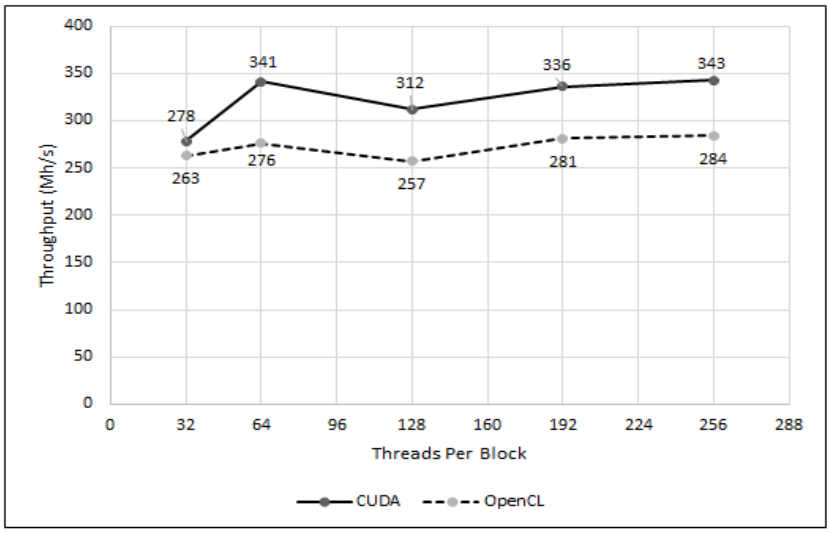

Figure 4.4: Depicts Mhashes/sec generated by varying threads per block from 32 to 256 during implementation of SHA-1 in CUDA and OpenCL

4.3 Results

4.3.1 Comparing Peak Performance

Calculate Performance Factor(Max) in DES Implementation

Maximum throughput by implementing DES in CUDA =78.11 Mkeys/sec. (from Fig.4.1)

Maximum throughput by implementing DES in OpenCL =58.45 Mkeys/sec. (from Fig.4.1)

$PF_{1Max.(DES)} = 78.11/58.45 = 1.33$

Where $PF_{1Max.(DES)}$ denotes the Performance Factor generated by implementation of DES in CUDA and OpenCL taking into account the maximum achievable throughput.

Calculate Performance Factor(Max) in MD5 Implementation

Maximum throughput by implementing MD5 in CUDA =2411 Mhashes/sec. (from Fig.4.2)

Maximum throughput by implementing MD5 in OpenCL =1891 Mhashes/sec. (from Fig.4.2)

$PF_{1Max.(MD5)} = 2411/1891 = 1.27$

Where $PF_{1Max.(MD5)}$ denotes the Performance Factor generated by implementation of MD5 in CUDA and OpenCL taking into account the maximum achievable throughput.

Calculate Performance Factor(Max) in SHA-1 Implementation

Maximum throughput by implementing SHA-1 in CUDA =343 Mhashes/sec. (from Fig.4.3)

Maximum throughput by implementing SHA-1 in OpenCL =284 Mhashes/sec. (from Fig.4.3)

$PF_{1Max.(MD5)}$= 343/284=1.20

Where $PF_{1Max.(SHA-1)}$ denotes the Performance Factor generated by implementation of SHA-1 in CUDA and OpenCL taking into account the maximum achievable throughput.

$PF_{1Max.}$ **= (1.33+1.27+1.20)/3=1.26.**

Where $PF_{1Max.}$ denotes the effective performance factor taking into account the maximum achievable throughput by implementation of pre-described algorithms.Fig.4.4 depicts the comparative results obtained.

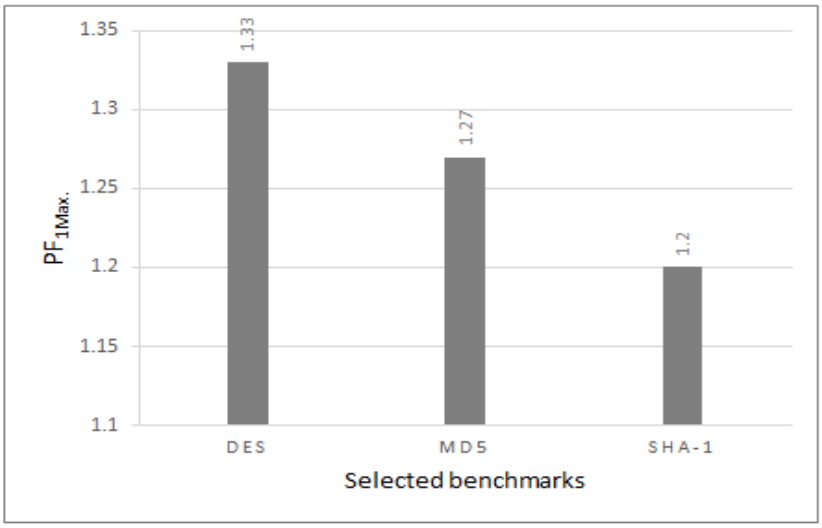

Figure 4.5: Comparing Peak Performance

4.3.2 Comparing Average Performance

Calculate Performance Factor(Avg.) in DES Implementation.

Average throughput by implementing DES in CUDA =73.44 Mkeys/sec. (from Fig.4.1)

Average throughput by implementing DES in OpenCL =51.08 Mkeys/sec. (from Fig.4.1)

$PF_{1Avg.(DES)}$= 73.44/51.08=1.43

Where $PF_{1Avg.(DES)}$ denotes the Performance Factor generated by implementation of DES in CUDA and OpenCL taking into account the average achievable throughput.

Calculate Performance Factor(Avg.) in MD5 Implementation

Average throughput by implementing MD5 in CUDA =2167.2 Mhashes/sec. (from Fig.4.2)

Average throughput by implementing MD5 in OpenCL =1783.2 Mhashes/sec. (from Fig.4.2)

$PF_{1Avg.(MD5)}$= 2167.2/1783.2=1.21

Where $PF_{1Avg.(MD5)}$ denotes the Performance Factor generated by implementation of MD5 in CUDA and OpenCL taking into account the average achievable throughput.

Calculate Performance Factor(Avg.) in SHA-256 Implementation

Average throughput by implementing SHA-1 in CUDA =322 Mhashes/sec. (from Fig.4.3)

Average throughput by implementing SHA-1 in OpenCL =272.2 Mhashes/sec. (from Fig.4.3)

$PF_{1Avg.(SHA-1)}$= 322/272.2=1.18

Where $PF_{1Avg.(SHA-1)}$ denotes the Performance Factor generated by implementation of SHA-256 in CUDA and OpenCL taking into account the average achievable throughput.

$PF_{1AVG.}$ = **(1.43+1.21+1.18)/3=1.27**

Where $PF_{1Avg.}$ denotes the effective performance factor taking into account the averaged achievable throughput by implementation of pre-described algorithms.Fig.4.5 depicts the comparative results obtained.

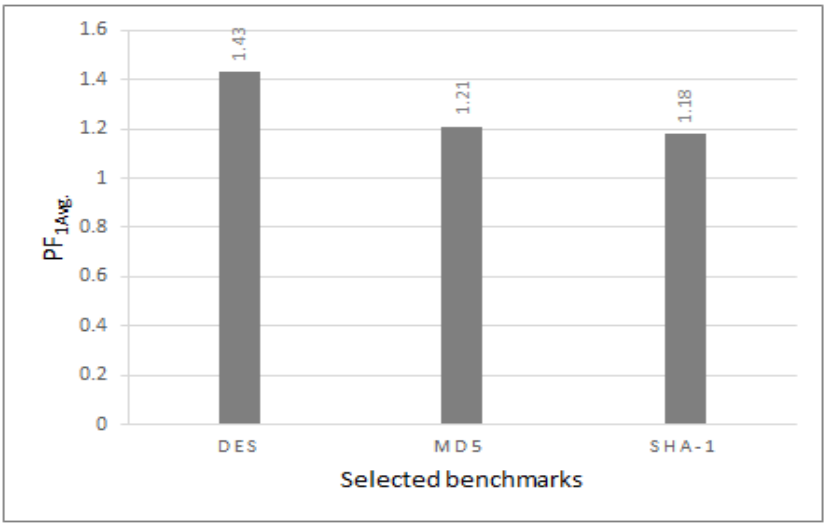

Figure 4.6: Comparing Average Performance

4.3.3 Comparing stability

Calculate performance factor (Stability) in DES Implementation.

Percentage Increase in throughput by varying threads per block from 64 to 256 by implementation of DES in CUDA=1.53. (from Fig. 4.1)

Percentage Increase in throughput by varying threads per block from 64 to 256 by implementation of DES in OpenCL=8.16. (from Fig. 4.1)

$PF_{2(DES)} = [1.53]^{-1}/[8.16]^{-1} = 5.33$

Where $PF_{2(DES)}$ denotes the Performance Factor generated by implementation of DES in CUDA and OpenCL w.r.t stability.

Calculate performance factor (Stability) in MD5 Implementation

Percentage Increase in throughput by varying threads per block from 64 to 256 by implementation of MD5 in CUDA=0.33. (from Fig. 4.2)

Percentage Increase in throughput by varying threads per block from 64 to 256 by implementation of MD5 in OpenCL=1.19. (from Fig. 4.2)

$PF_{2(MD5)} = [0.33]^{-1}/[1.19]^{-1} = 3.6$

Where $PF_{2(MD5)}$ denotes the Performance Factor generated by implementation of MD5 in CUDA and OpenCL w.r.t stability.

Calculate performance factor (Stability) in SHA-1 Implementation

Percentage Increase in throughput by varying threads per block from 64 to 256 by implementation of SHA-1 in CUDA=9.93. (from Fig. 4.3)

Perentage Increase in throughput by varying threads per block from 64 to 256 by implementation of SHA-1 in OpenCL=7.98. (from Fig. 4.3)

$PF_{2(SHA-256)} = [0.58]^{-1}/[2.81]^{-1} = 4.84$

Where $PF_{2(SHA-256)}$ denotes the Performance Factor generated by implementation of SHA-1 in CUDA and OpenCL w.r.t stability.

PF_2 **= (5.33+3.6+4.84)/3=4.59**

Where PF_2 is the effective Performance factor taking into account the achievable stability by implementation of DES, MD5, and SHA-1. Fig. 4.6 depicts the results so obtained.

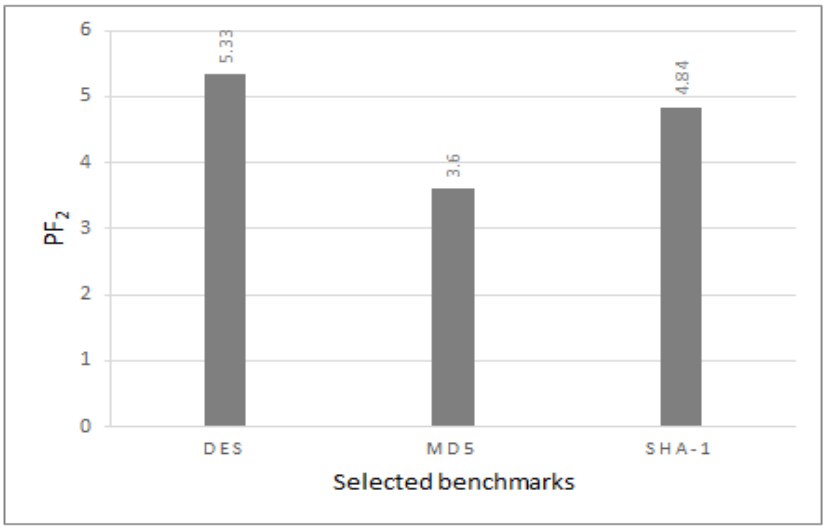

Figure 4.7: Comparing Stability

Chapter 5

Conclusions

In this paper three different algorithms, i.e. DES, MD5 and SHA-1 have been used to compare the performance of CUDA with NVIDIAs implementation of OpenCL. In our tests, CUDA scored over OpenCL in terms of Peak Performance and Average Performance respectively. This is deducible from the fact that, on the average scenario, CUDA performs 27 percent better than OpenCL while in the best case scenario it takes over OpenCL by 30 percent.

As far as the optimal number of threads per block goes, 256 threads per block is the most performant choice, proving that the CUDA architecture is able to deal with an increased pressure on the register file without problems as CUDA scores 4.5times over OpenCL in terms of stability.

Thus, CUDA seems to be a better choice for applications where achieving as high a performance as possible is the main priority. Otherwise, the choice between CUDA and OpenCL can be made by taking into account factors such as prior familiarity with either system, or available development tools for the target GPU hardware.

Bibliography

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, pp. 80–113, Wiley Online Library, 2007.

[2] S. Menon, "An extensible automated performance-tuning framework for gpu based applications,"

[3] A. Raina, *Analysis of Parallel Sorting Algorithms on Heterogeneous Processors with OpenCL*. PhD thesis, National Institute of Technology Rourkela, 2013.

[4] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005.

[5] S. William and W. Stallings, *Cryptography and Network Security, 4/E*. Pearson Education India, 2006.

[6] J. Agrawal, S. Shahu, *et al.*, "New modified 256-bit md5 algorithm with sha compression function," *International Journal of Computer Applications*, vol. 42, no. 12, pp. 47–51, 2012.

[7] P. Gupta and S. Kumar, "A comparative analysis of sha and md5 algorithm," *architecture*, vol. 1, p. 5, 2014.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance eval-

uation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, ACM, 2008.

[9] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Gpu-assisted malware," *International Journal of Information Security*, pp. 1–9, 2010.

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[11] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, pp. 80–113, Wiley Online Library, 2007.

[12] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE micro*, no. 4, pp. 13–27, 2008.

[13] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[14] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[15] J. McDonald Jr and B. Burley, "Per-face texture mapping for real-time rendering," in *ACM SIGGRAPH 2011 Studio Talks*, p. 3, ACM, 2011.

[16] R. J. Rost, B. M. Licea-Kane, D. Ginsburg, J. M. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen, *OpenGL shading language*. Pearson Education, 2009.

[17] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa, "The opencl programming book," *Fixstars Corporation*, vol. 63, 2010.

[18] R. M. Baecker, *Readings in groupware and computer-supported cooperative work: Assisting human-human collaboration*. Morgan Kaufmann, 1993.

[19] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[20] G. Agosta, A. Barenghi, F. De Santis, and G. Pelosi, "Record setting software implementation of des using cuda," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pp. 748–755, IEEE, 2010.

[21] T. Nie and T. Zhang, "A study of des and blowfish encryption algorithm," in *TENCON 2009-2009 IEEE Region 10 Conference*, pp. 1–4, IEEE, 2009.

[22] H. W. Van Rumpt, J. Van Maanen, N. J. Opdam, and W. J. Vervoorn, "Device for enciphering and deciphering, by means of the des algorithm, data to be written to be read from a hard disk," Apr. 30 1996. US Patent 5,513,262.

[23] R. Rivest, "The md5 message-digest algorithm," 1992.