
IMPLEMENTATION OF SECURITY ALGORITHM USING CUDA

By
Snehal K. Ambulkar
08MCES51



Department of
Computer Science and Engineering
Ahmedabad-382481

May 2011

**IMPLEMENTATION OF SECURITY ALGORITHM
USING CUDA**

Major Project

**Submitted in partial fulfillment of the requirements
For the degree of
Master of Technology in Computer Science and Engineering**

By

Snehal K. Ambulkar

Guided By

Prof. Samir Patel



**Department of
Computer Science and Engineering
Ahmedabad-382481**

Declaration

This is to certify that

- i) The thesis comprises my original work towards the degree of Master of Technology in Computer Science and Engineering at Nirma University and has not been submitted elsewhere for a degree.
- ii) Due acknowledgement has been made in the text to all other material used.

Snehal K. Ambulkar

Certificate

This is to certify that the Major Project entitled “**Implementation Of Security Algorithm Using CUDA**” submitted by **Snehal K. Ambulkar (08MCES51)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University, Ahmedabad is the record of work carried out by her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven’t been submitted to any other university or institution for award of any degree or diploma.

Prof. Samir Patel

Guide, Senior Associate Professor,

Department of CSE,

Nirma University, Ahmedabad.

Dr. S.N.Pradhan

Professor and PG-Coordinator,

Department of CSE,

Nirma University, Ahmedabad

Prof. D.J. Patel

Professor & Head,

Department of CSE,

Nirma University, Ahmedabad.

Dr. K.Kotecha

Director,

Nirma University, Ahmedabad.

Abstract

Multicore processors are now present in many home based systems. This chance presents a massive challenge to application developers who must design a sufficient and suited parallelism onto each parallel algorithm. Considering today's hardware performance, in order to obtain best results, a proper programming strategy for optimum mapping of all processes to existing resources is necessary. The presence of multiple cores in a single chip requires applications with a higher level of parallelism. The use of suitable mapping algorithms can lead to a great performance improvement considering computing time at smaller energy consumption. Mapping a set of algorithms onto a multi core platform requires using a parallel programming model, which describes and controls the communication, concurrences, and synchronization of all components involved. Basic goal of CUDA is to help programmers focus on the task of parallelization of the algorithms.

Acknowledgements

My deep sense of gratitude to **Prof. Samir Patel**, Senior Associate Professor for his continual encouragement throughout the Major project. Being my project guide, he has taken the pain to go through the project.

My deepest thank to **Dr. S.N.Pradhan**, Professor and PG-Coordinator of Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for giving his valuable inputs and correcting various documents of mine with attention and care.

I would like to thank **Dr. Ketan Kotecha**, Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad and **Prof. D.J. Patel** Head of Department, for their unmentionable support, providing basic infrastructure and healthy research environment.

I would also like to thank to all my colleagues and friends and those who are directly or indirectly involved in this project without whom this project would have been a distant reality. Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete my dissertation work successfully.

Finally, I would like to thank GOD for his blessings.

- **Snehal K. Ambulkar**

Contents

Declaration	ii
Certificate	iii
Abstract	iv
Acknowledgements	v
List of Figures	xi
List of Tables	xii
1 Introduction	1
1.1 General Overview	1
1.2 Motivation	2
1.3 Objective	2
1.4 System Requirements	3
1.5 Technical Specifications	3
1.6 Thesis Organization	3
2 Literature Survey	5
2.1 CPU and GPU Comparison	5
2.2 CUDA Introduction	6
2.3 Advantages	6

2.4	Limitations	7
2.5	Current & Future use of CUDA Architecture	7
2.6	CUDA Processing Flow	8
2.7	CUDA Programming Model	9
2.7.1	Function Qualifiers	10
2.7.2	Built-In Variables	11
2.8	CUDA Memory Model	12
2.9	GPU/CPU Synchronization	14
2.10	Memory Optimization	15
2.10.1	Host-Device Data Transfers	15
2.11	CUDA Event API	16
2.12	NVIDIA GeForce G105M	16
2.12.1	Features	17
3	Tools and Techniques	18
3.1	CUDA installation	18
3.2	CUDA Compiler - NVCC	19
3.3	CUDA Occupancy Calculator	19
3.3.1	Role Of CUDA compiler	20
3.3.2	Limitation	20
3.3.3	Notes about Occupancy	21
3.3.4	Kernel Usage	21
3.4	NVIDIA Compute Visual Profiler	22
3.4.1	Compute program	22
3.5	Microsoft Visual Studio 2008	23
3.5.1	Specific Steps	24
3.6	Sample Project	25
3.6.1	main.cpp	25
3.6.2	KernelWraper.cu	26

3.6.3	MyKernel.cu	27
4	Data Encryption Standard	28
4.1	Introduction	28
4.2	Modes of Operation for Encryption Algorithms	30
4.2.1	Block Cipher	30
4.2.2	Stream Cipher	30
4.3	Modes of Operation for Block Ciphers	32
4.3.1	ECB (Electronic Code Book)	32
4.3.2	CBC (Cipher Block Chaining)	33
4.3.3	CFB (Cipher Feedback)	33
4.3.4	OFB (Output Feedback)	34
4.4	Conventional DES Implementation	35
5	Implementation	39
5.1	CUDA-DES algorithm implementation using Proposed Model	40
5.1.1	Theoretical Analysis for Security Algorithm	40
5.1.2	Amdahl's law :	41
5.1.3	CUDA Program Flow :	42
6	Result and Analysis	45
6.1	Experiment 1	45
6.2	Experiment 2	46
6.3	Experiment 3	47
6.4	Result Analysis	49
7	Conclusion and Future Work	52
7.1	Conclusion	52
7.2	Future Work	53
	Web References	54

<i>CONTENTS</i>	ix
Bibliography	56
Index	57

List of Figures

2.1	CUDA Architecture	6
2.2	Processing Flow	8
2.3	CUDA Programming Model	9
2.4	Memory Space	12
2.5	A set of SIMD multiprocessor with on-chip shared memory	13
3.1	Threads per Block	20
3.2	Custom Build Rule	24
3.3	Property page	25
4.1	Main parts of DES Algorithm	29
4.2	Block Cipher	30
4.3	Stream Cipher	31
4.4	Electronic Code Book	32
4.5	Cipher Block Chaining	33
4.6	Cipher Feedback	34
4.7	Output Feedback	35
4.8	DES Architecture	37
4.9	Single round of DES algorithm	38
4.10	DES Architecture for one round	38
5.1	Proposed Model	39
5.2	Amdahl's Law for Speedup	41

5.3	CUDA Flow	43
6.1	CPU Encryption	46
6.2	GPU Encryption	46
6.3	Encryption for 16 round DES on CPU	47
6.4	Decryption for 16 round DES on CPU	47
6.5	Encryption for 16 round DES on CPU + GPU	49
6.6	Decryption for 16 round DES on CPU + GPU	49
6.7	CPU Encryption using file	50
6.8	CPU + GPU Encryption using file	51
6.9	Results for Encryption in terms of time	51

List of Tables

I	Results for Encryption	45
II	Results for Encryption and Decryption on CPU	48
III	Results for Encryption and Decryption on CPU + GPU	48
IV	Results for Encryption in terms of time in seconds	48

Chapter 1

Introduction

1.1 General Overview

According to websters, Architecture is an “Art or Science of building; a method or style of building”. Computer architecture comprises both the art and science of designing new and faster computer systems to satisfy ever increasing demand for more powerful systems. A computer architect specifies the modules that form the computer system at a functional level of detail and also specifies the interfaces between these modules. The exact mix of hardware, software and firmware used to implement the module depends on performance requirements, cost and availability of hardware, software and firmware.

Performance and cost are the two major parameters for evaluation of architecture. Here, the aim is to maximize performance while minimizing the cost(i.e. to maximize the performance-to-cost ratio). The progress in technologies provides new choices each year. The architect has not only base his or her decisions on the choices available today, but also keep in mind the expected changes in technology during life of system. The architectural feature that provides the optimum performance-to-cost ratio today may not be the best feature for tomorrow’s technology.

1.2 Motivation

Multicore [12](dual, quad etc) processors are now present in many home based systems. This chance presents a massive challenge to application developers who must design a sufficient and suited parallelism onto each parallel algorithm. Mapping a set of algorithms onto a multi core platform requires using a parallel programming model, which describes and controls the communication, concurrences, and synchronization of all components involved. The correct use of synchronization and locking mechanisms is complex and it has proven to be a challenging implementation.

It is often said that limits of semiconductor technology have already been reached and no significant performance improvements are possible by technology improvements alone. History has always proven this statement to be false. But we always make the best out of the existing technology. The alternative architectures utilize the parallel and overlapped(pipelined)processing possibilities.Parallel processing is considered as the fourth wave of computing. Parallel processing architectures utilize a multiplicity of processors and provide for building computer system with order of magnitude performance increase. In order to utilize parallel processing architectures efficiently,an extensive redesign of algorithms and data structures is needed.

Using CUDA, the latest NVIDIA GPUs[10][13] become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very fast.

1.3 Objective

Objective of this thesis is to implement complex Data Encryption Standard (DES) algorithm using CUDA environment so as to study relative performance of CPUs and GPUs.

1.4 System Requirements

To use CUDA on your system, you will need the following installed:

- CUDA-enabled GPU(here, GeForce G105M)
- Device driver[4]
- CUDA software (available at no cost from <http://www.nvidia.com/cuda>)
- Microsoft Visual Studio .NET 2003, Microsoft Visual Studio 2008, or the corresponding versions of Microsoft Visual C++ Express Edition.
- NVCC Compiler.[3]

1.5 Technical Specifications

Technical Specification of GeForce G105M GPU Device:

- Total amount of global memory: 521601024 bytes
- Number of multiprocessors: 1
- Number of cores: 8
- Total amount of constant memory: 65536 bytes
- Total amount of shared memory per block: 16384 bytes
- Total number of registers available per block: 8192
- Warp size: 32
- Maximum number of threads per block: 512
- Maximum size of each dimension of a block: 512 x 512 x 64
- Maximum size of each dimension of a grid: 65535 x 65535 x 1
- Clock rate: 0.78 GHZ

1.6 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Literature survey on CUDA describes CPU & GPU. It also describes*

CUDA Programming Model ,Memory Architecture and various memory optimization techniques.

Chapter 3, *Here, how to use CUDA Technology and various tools is described.*

Chapter 4, *Various modes of encryption/decryption is described here.*

Chapter 5, *This chapter describes algorithm implementation using CUDA and theoretical analysis for speedup.*

Chapter 6, *describes results and analysis of results.*

Chapter 7, *Conclusion and Future work is given in this chapter.*

Chapter 2

Literature Survey

2.1 CPU and GPU Comparison

- CPU cores are designed to execute a single thread of sequential instructions with maximum speed and GPUs are designed for fast execution of many parallel instruction threads.
- CPUs use SIMD and GPUs use SIMT.
- GPUs contain much larger number of dedicated ALUs than CPUs.
- CPUs reduce memory access latencies using large caches as well as branching prediction. GPUs solve the problem of memory access latencies using simultaneous execution of thousands of threads when one thread is waiting for data from memory, a GPU can execute another thread without latencies.
- CPUs use caches to increase their performance owing to reduced memory access latencies and GPUs use caches or shared memory to increase memory bandwidth.

2.2 CUDA Introduction

CUDA (an acronym for Compute Unified Device Architecture)[1] is a parallel computing architecture developed by NVIDIA[8]. CUDA is the computing engine in NVIDIA GPUs(Graphics Processing Units) that is accessible to software developers through variants of industry standard programming languages. Figure 2.1 shows the

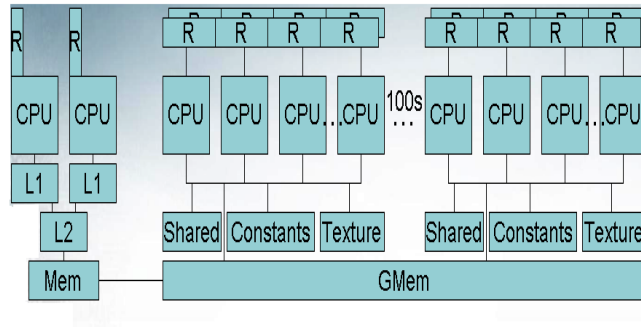


Figure 2.1: CUDA Architecture

CUDA architecture. Using CUDA, the latest NVIDIA GPUs become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very fast. This approach of solving general purpose problems on GPUs is known as GPGPU[4]. CUDA works with all NVIDIA GPUs from the G8X series onwards, including GeForce, Quadro and the Tesla line.

2.3 Advantages

CUDA has several advantages over traditional general purpose computation on GPUs (GPGPU) using graphics APIs[9].

- Scattered reads - code can read from arbitrary addresses in memory.

- Shared memory - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU.
- Full support for integer and bitwise operations, including integer texture lookups.

2.4 Limitations

- CUDA uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
- Texture rendering is not supported.
- The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD [11] execution model becomes a significant limitation for any inherently divergent task.

2.5 Current & Future use of CUDA Architecture

Some of the current and future usages of CUDA are as follows:

- Accelerated rendering of 3D graphics
- Real Time Cloth Simulation
- Distributed Calculations, such as predicting the native conformation of proteins

- Medical analysis simulations, for example virtual reality based on CT and MRI scan images
- Physical simulations, in particular in fluid dynamics
- Accelerated encryption, decryption and compression
- Artificial intelligence etc.

2.6 CUDA Processing Flow

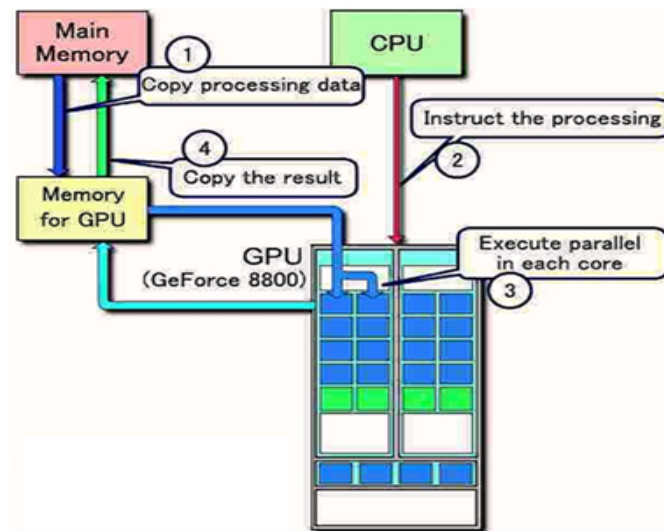


Figure 2.2: Processing Flow

Figure 2.2 shows CUDA processing flow. The steps for processing flow are as follows:

1. Copy data from main memory to GPU memory
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU memory to main memory

2.7 CUDA Programming Model

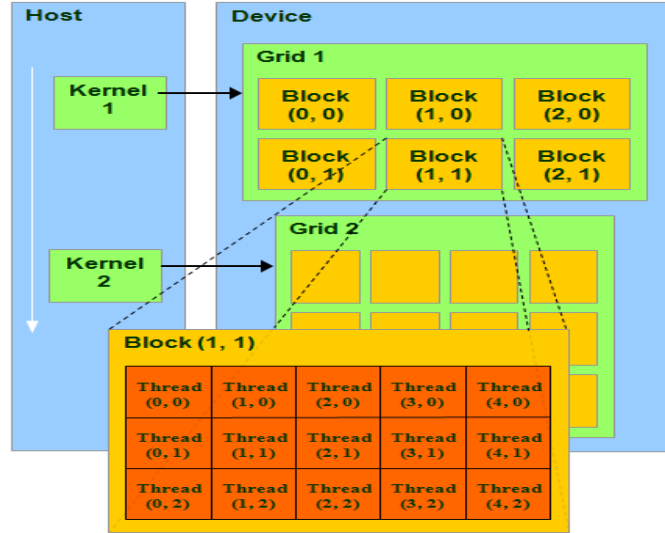


Figure 2.3: CUDA Programming Model

The Compute Unified Device Architecture, proposed by NVIDIA for its G80, G92 and GT200 etc. graphics processors, exposes a programming model that integrates host and GPU code in the same C++ source files. The main programming introduced by the programming model is an explicitly parallel function invocation (kernel) which is executed by a user-specified number of threads. Every CUDA kernel is explicitly invoked by host code and executed by the device, while the host-side code continues the execution asynchronously after instantiating the kernel. The programmer is provided with a specific synchronizing function call to wait for the completion of the active asynchronous kernel computation.

The CUDA programming model abstracts the actual parallelism implemented by the hardware architecture, providing the concepts of block and thread to express concurrency in algorithms. A block captures the notion of a group of concurrent threads. Blocks are required to execute independently, so that it has to be possible to execute them in any order (in parallel or in sequence). Therefore, the synchronization primitives semantically act only among threads belonging to the same block. Intra-block

communications among threads use the logical shared memory associated with that block. Since the architecture does not provide support for message-passing, threads belonging to different blocks must communicate through global memory which is entirely mapped to the on-chip memory. The concurrent accesses to logical shared memory by threads executing within the same block are supported through an explicit barrier synchronization primitive.

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. CUDA uses parallel computing model, when each of SIMD [11] processors executes the same instruction over different data elements in parallel. GPU is a computing device, co-processor (device) for a CPU (host), possessing its own memory and processing a lot of threads in parallel. A kernel is a GPU function executed by threads.[6] Figure 2.3 shows that the host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks.

Thread blocks are executed in the form of small groups called warps (32 threads each). It is minimum volume of data, which can be processed by multiprocessors. As its not always convenient, CUDA allows to work with blocks containing 64 - 512 threads.

Grouping blocks into grids helps avoid the limitations and apply the kernel to more threads per call. It also helps in scaling. If a GPU does not have enough resources, it will execute blocks one by one. Otherwise, blocks can be executed in parallel, which is important for optimal distribution of the load on GPUs of different levels, starting from mobile and integrated solutions.[8]

2.7.1 Function Qualifiers

- `__device__` The device qualifier declares a function that is: - Executed on the device. - Callable from the device only.
- `__global__` The global qualifier declares a function that is: - Executed on the device. - Callable from the host only.

- `__host__` The host qualifier declares a function that is:
 - Executed on the host.
 - Callable from the host only.

It is equivalent to declare a function with only the host qualifier or to declare it without any of the host , device , or global qualifier; in either case the function is compiled for the host only. However, the host qualifier can also be used in combination with the device qualifier, in which case the function is compiled for both the host and the device.

- Restrictions
 - 1) `__device__` and `__global__` functions do not support recursion.
 - 2) `__device__` and `__global__` functions cannot declare static variables inside their body.
 - 3) `__device__` and `__global__` functions cannot have a variable number of arguments.
 - 4) `__device__` functions cannot have their address taken; function pointers to global functions, on the other hand, are supported.
 - 5) The global and host qualifiers cannot be used together.
 - 6) `__global__` functions must have void return type.
 - 7) Any call to a global function must specify its execution configuration.
 - 8) A call to a `__global__` function is asynchronous, meaning it returns before the device has completed its execution.
 - 9) `__global__` function parameters are currently passed via shared memory to the device and limited to 256 bytes.

2.7.2 Built-In Variables

- `gridDim` - This variable is of type `dim3` and contains the dimensions of the grid.
- `blockIdx` - This variable is of type `uint3` and contains the block index within the grid.

- `blockDim` - This variable is of type `dim3` and contains the dimensions of the block.
- `threadIdx` - This variable is of type `uint3` and contains the thread index within the block.
- `WarpSize` - This variable is of type `int` and contains the warp size in threads.
- `Restrictions` - It is not allowed to take the address of any of the built-in variables and to assign values to any of the built-in variables.

2.8 CUDA Memory Model

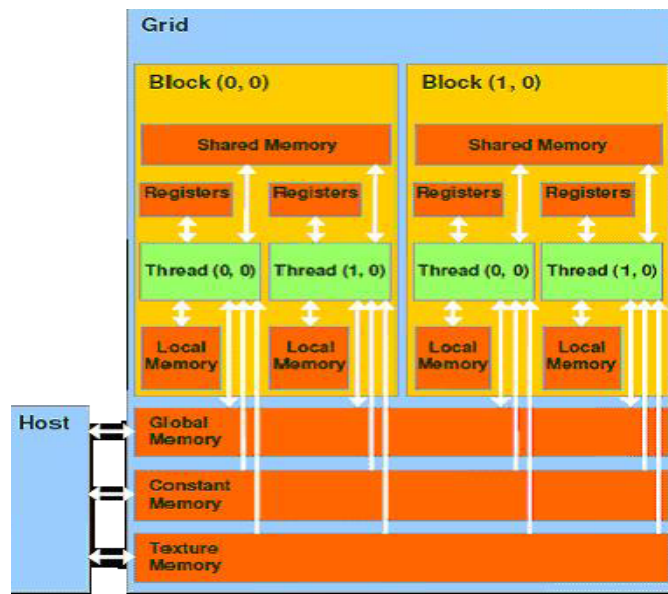


Figure 2.4: Memory Space

CUDA memory model[9] allows byte wise addressing, support for gather and scatter. There are quite a lot of registers per each streaming processor, up to 1024. Access to these registers is very fast, they can store 32-bit integer or floating point numbers. Each thread has access to the following memory types: Figure 2.4 shows that thread

has access to the devices DRAM and on-chip memory through a set of memory spaces of various scopes.

- Global memory : the largest volume of memory available to all multiprocessors in a GPU, from 256 MB to 1.5 GB in modern solutions (and up to 4 GB in Tesla). It offers high bandwidth, over 100 GB/s for top solutions from NVIDIA, but it suffers from very high latencies (several hundred cycles). Non-cacheable, supports general load and store instructions, and usual pointers to memory.
- Local memory : small volume of memory, which can be accessed only by one streaming processor. It's relatively slow, just like global memory.

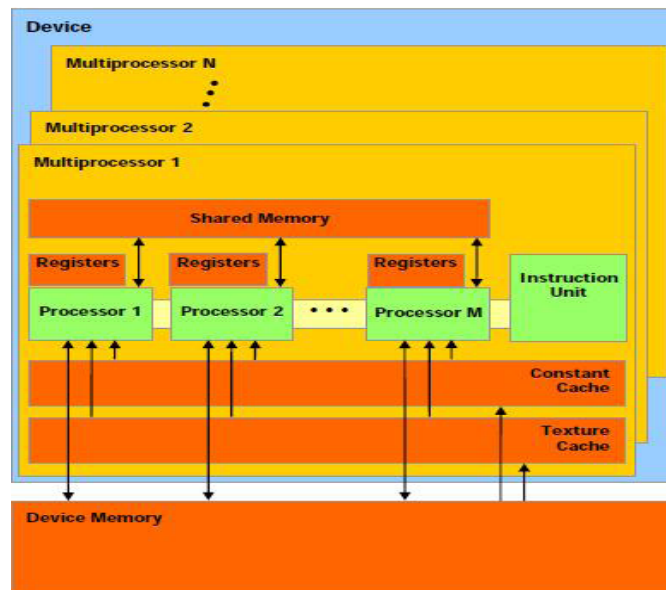


Figure 2.5: A set of SIMD multiprocessor with on-chip shared memory

- Shared memory : 16-KB memory (in graphics processors of the state-of-the-art architecture) shared between all streaming processors in a multiprocessor. It's fast memory, just like registers. This memory provides interaction between threads, it's controlled by developers directly and features low latencies. Advantages of shared memory: it can be used as a controllable L1 Cache, reduced

latencies when ALUs access data, fewer calls to global memory. Figure 2.5 shows a set of SIMD multiprocessor with on-chip shared memory.

- Constant storage - memory area of 64 KB (the same concerns modern GPUs), read only for all multiprocessors. It's cached by 8 KB for each multiprocessor. This memory is rather slow latencies of several hundred cycles, if there are no required data in cache.
- Texture memory : is available for reading to all multiprocessors. Data are fetched by texture units in a GPU, so the data can be interpolated linearly without extra overheads. Cached by 8 KB for each multiprocessor. Slow as global memory latencies of several hundred cycles, if there are no required data in cache.

It goes without saying that global, local, texture, and constant memory is physically the same memory. They differ only in caching algorithms and access models. CPU can refresh and access only external memory: global, constant, and texture memory.

2.9 GPU/CPU Synchronization

- Context based
`cudaThreadSynchronize()` - Blocks until all previously issued CUDA calls from a CPU thread complete.
- Stream based
`cudaStreamSynchronize(stream)` - Blocks until all CUDA calls issued to given stream complete.
`cudaStreamQuery(stream)`
 - Indicates whether stream is idle
 - Returns `cudaSuccess`, `cudaErrorNotReady`, ...
 - Does not block CPU thread

- Stream based using events

Events can be inserted into streams: `cudaEventRecord(event, stream)`

Event is recorded then GPU reaches it in a stream. Recorded = assigned a timestamp (GPU clocktick).

`cudaEventSynchronize(event)` - Blocks until given event is recorded.

`cudaEventQuery(event)`

- Indicates whether event has recorded.
- Returns `cudaSuccess`, `cudaErrorNotReady`
- Does not block CPU thread.

2.10 Memory Optimization

There are two types of memory optimizations:

1. Data transfers between host and device.
2. Device memory optimizations.

2.10.1 Host-Device Data Transfers

Device to host memory bandwidth much lower than device to device bandwidth. It has a 8 GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280). In minimize transfers intermediate data can be allocated, operated on and deallocated without ever copying them to host memory. In group transfers one large transfer much better than many small ones.

Host Synchronization

- All kernel launches are asynchronous: control returns to CPU immediately and kernel executes after all previous CUDA calls have completed.
- `cudaMemcpy()` is synchronous control returns to CPU after copy completes and copy starts after all previous CUDA calls have completed.

- `cudaThreadSynchronize()` blocks until all previous CUDA calls complete.
- Async API provides GPU CUDA - call streams non-blocking `cudaMemcpyAsync`.

2.11 CUDA Event API

- Events are inserted (recorded) into CUDA call streams.

- Usage scenarios:

- measure elapsed time for CUDA calls(clock cycle precision)
- query the status of an asynchronous CUDA call
- block CPU until CUDA calls prior to the event are completed

- AsyncAPI sample in CUDA SDK:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel(((grid,block)))(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

2.12 NVIDIA GeForce G105M

The Nvidia GeForce G105M is a higher clocked GeForce 9300M G and therefore a DirectX 10 capable graphics adapter for small and thin notebooks. Compared to the

9300M G, the GeForce G105M features only 8. To overcome this downside, it features a higher clock speed. Furthermore, it integrates the new VP3 video processor (8400M, 9300M G had VP2) with more video features and better hd-video decoding support. GeForce G105M GPUs play all the latest games with all the detail, lighting and special effects the way they were meant to be played. The Pure Video HD processor plays full HD and Blu-ray movies with cinematic quality and battery sipping power. The included CUDA technology improves video and photo editing productivity by handling tasks up to 5 times faster than mainstream CPUs.

2.12.1 Features

- High Performance GeForce DirectX 10 Graphics Processor GeForce G105M includes a powerful DirectX 10, Shader Model 4.0 graphics processor, offering full compatibility with past and current game titles with all the texture detail, high dynamic range lighting and visual special effects the game developer intended the consumer to see.
- CUDA technology NVIDIA CUDA technology unlocks the power of the GPU's processing cores to accelerate the most demanding system tasks - such as video encoding - delivering up to 5x the performance over traditional notebook CPUs for applications that support CUDA technology.
- NVIDIA PureVideo HD technology The combination of high-definition video decode acceleration and post-processing that delivers unprecedented picture clarity, smooth video, accurate color, and precise image scaling for movies and video. Feature requires supported video software. Feature may vary by product.
- NVIDIA Hybrid SLI - Hybrid Power technology Hybrid Power technology intelligently powers up the GeForce G105M for optimal performance when you are plugged in and turns it off when you unplug to extend battery.

Chapter 3

Tools and Techniques

3.1 CUDA installation

CUDA installation[2] consists of:

- a. Driver : Required component to run CUDA applications.
- b. CUDA Toolkit (compiler,libraries)
- c. CUDA SDK[7](example codes)

The CUDA Toolkit is a C language development environment for CUDA-enabled GPUs. The CUDA development environment includes:

- NVCC compiler
- CUDA libraries
- CUDA runtime driver is included into the standard NVIDIA driver
- Profiler

The CUDA Developer SDK provides examples with source code to help you get started with CUDA. Examples include:

- Matrix multiplication

- Thread Migration

CUDA includes C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware from developers. Although CUDA requires programmers to write special code for parallel processing, it does not require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU.

3.2 CUDA Compiler - NVCC

- Any source file containing CUDA language extensions (.cu) must be compiled with `nvcc`.
- NVCC is a compiler driver works by invoking all the necessary tools and compilers like `cudacc`, `g++`, `cl`,. . . .
- NVCC can output:
 - Either C code (CPU Code)- To be compiled with the rest of the application using another tool.
 - Or PTX object code directly.
- An executable with CUDA code requires: - The CUDA core library (`cuda`) - The CUDA runtime library (`cudart`)

3.3 CUDA Occupancy Calculator

The CUDA Occupancy Calculator ,as shown in figure 3.1allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported

on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA program threads. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor. The size of N on

- GPUs with compute capability 1.0-1.1 is $N = 8192$
- GPUs with compute capability 1.2-1.3, is $N = 16384$.
- GPUs with compute capability 2.0, is $N = 32768$.

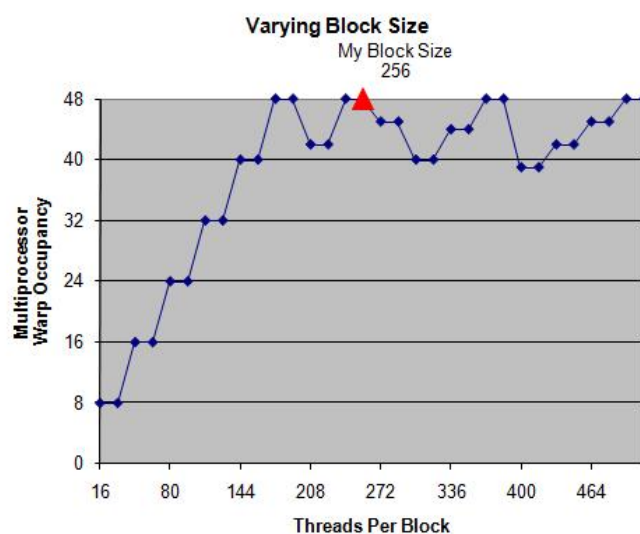


Figure 3.1: Threads per Block

3.3.1 Role Of CUDA compiler

The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously.

3.3.2 Limitation

If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N , the launch will fail.

Maximizing the occupancy can help to cover latency during global memory loads that are followed by `a_syncthreads()`. The occupancy is determined by the amount of shared memory and registers used by each thread block. Because of this, programmers need to choose the size of thread blocks with care in order to maximize occupancy. This GPU Occupancy Calculator can assist in choosing thread block size based on shared memory and register requirements.

3.3.3 Notes about Occupancy

Higher occupancy does not necessarily mean higher performance. If a kernel is not bandwidth bound, then increasing occupancy will not necessarily increase performance. If a kernel invocation is already running at least one thread block per multiprocessor in the GPU, and it is bottlenecked by computation and not by global memory accesses, then increasing occupancy may have no effect. In fact, making changes just to increase occupancy can have other effects, such as additional instructions, spills to local memory (which is off chip), divergent branches, etc. For bandwidth-bound applications, on the other hand, increasing occupancy can help better hide the latency of memory accesses, and therefore improve performance.

3.3.4 Kernel Usage

To determine the number of registers used per thread in your kernel, simply compile the kernel code using the option `-ptxas-options = -v` to `nvcc`. This will output information about :

- 1) Register,
- 2) Local memory,
- 3) Shared memory, and
- 4) Constant memory

usage for each kernel in the `.cu` file. However, if your kernel declares any external shared memory that is allocated dynamically, you will need to add the (statically

allocated) shared memory reported by ptxas to the amount you dynamically allocate at run time to get the correct shared memory usage. An example of the verbose ptxas output is as follows: ptxas info : Compiling entry function ‘_Z8my_kernelPf’ for ‘sm_10’ ptxas info : Used 5 registers, 8+16 bytes smem Let’s say “my_kernel” contains an external shared memory array which is allocated to be 2048 bytes at run time. Then our total shared memory usage per block is $2048+8+16 = 2072$ bytes. We enter this into the box labeled “shared memory per block (bytes)” in this occupancy calculator, and we also enter the number of registers used by my_kernel, 5, in the box labeled registers per thread. We then enter our thread block size and the calculator will display the occupancy.

3.4 NVIDIA Compute Visual Profiler

3.4.1 Compute program

Execute a CUDA or OpenCL program (referred to as Compute program in this document) with profiling enabled and view the profiler output as a table. The table has the following columns for each GPU method:

- GPU Timestamp : GPU start time stamp in micro seconds.
- Method : GPU method name. This is either “memcpy*” for memory copies or the name of a GPU kernel. Memory copies have a suffix that describes the type of a memory transfer, e.g. “memcpyDToHasync” means an asynchronous transfer from Device memory to Host memory.
- GPU Time : It is the execution time for the method on GPU.
- CPU Time : It is sum of GPU time and CPU overhead to launch that Method. At driver generated data level, CPU Time is only CPU overhead to launch the Method for non-blocking Methods; for blocking methods it is sum of GPU time and CPU overhead. All kernel launches by default are non-blocking. But if

any profiler counters are enabled kernel launches are blocking. Asynchronous memory copy requests in different streams are non-blocking.

- Stream Id : Identification number for the stream
- Columns only for kernel methods :
 - Occupancy : Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of active warps.
 - Profiler counters: Refer the profiler counters section for list of counters supported.
 - grid size X : Number of blocks in the grid along dimension X
 - grid size Y : Number of blocks in the grid along dimension Y
 - block size X : Number of threads in a block along dimension X
 - block size Y : Number of threads in a block along dimension Y
 - block size Z : Number of threads in a block along dimension Z
 - dyn smem per block : Dynamic shared memory size per block in bytes
 - sta smem per block : Static shared memory size per block in bytes
 - reg per thread : Number of registers per thread
- Columns only for memcpy methods :
 - mem transfer size : Memory transfer size in bytes
 - hostmem transfer type : Type of host memory during transfer from DtoH or HtoD. It can be either pageable or pagelocked.

3.5 Microsoft Visual Studio 2008

As far as implementation of Security Algorithm is concerned, it is very important to have the knowledge of development environment. So, this phase of project development includes:

- a. Study of Visual Studio.
- b. Building Sample Project in Visual Studio Environment.
- c. Integration of .cu and .cpp files in a project.

3.5.1 Specific Steps

One should use following steps for creating new project in visual studio environment:

- a. Create a new project using the standard MS wizards (e.g. an empty console project).
- b. Implement your host (serial) code in .c or .cpp files.
- c. Implement your wrappers and kernels in .cu files.
- d. Add the NvCudaRuntimeApi.rules (right click on the project, Custom Build Rules(see Figures 3.2 and 3.3), tick the relevant box).
- e. Add the CUDA runtime library.
- f. Add the CUDA include files.
- g. Then just build your project and the .cu files will be compiled to .obj and added to the link automatically

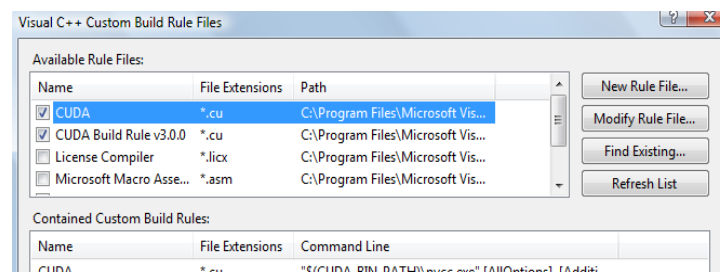


Figure 3.2: Custom Build Rule

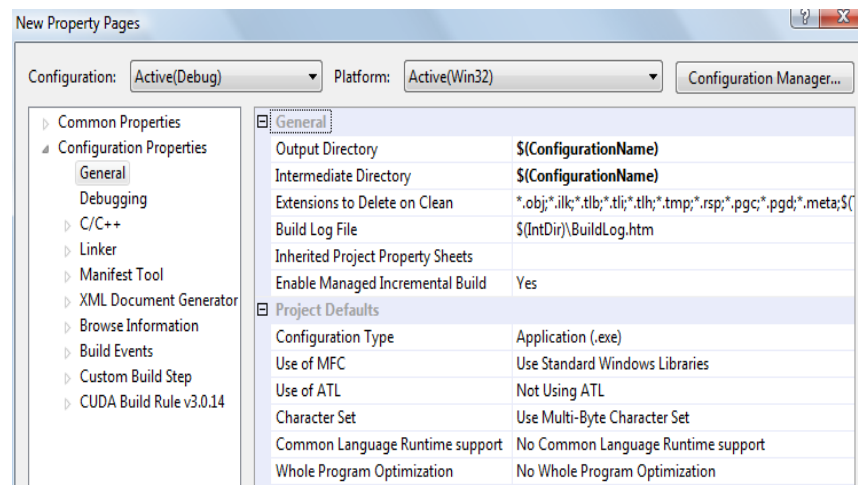


Figure 3.3: Property page

3.6 Sample Project

Here the sample project called “New Project” is given, which includes three source files as:

- 1) main.cpp
- 2) KernelWrapper.cu
- 3) MyKernel.cu

Here, main.cpp file contains wrapper function which is available in KernelWrapper.cu file. Now, KernelWrapper.cu file contains `_global__kernel` function which in turn is available in MyKernel.cu file.

Hence, in this way wrapper function is important for CUDA program development in visual studio environment.

3.6.1 main.cpp

```
#include <iostream >
#include<conio.h >
void RunTest();
```

```
//forward declaration
int main( int argc, char** argv)
{
    RunTest();
    std::cout << "It is working Fine...\n";
    getch();
    return 0;
}
```

3.6.2 KernelWraper.cu

```
#include <iostream >
__global__ void TestDevice(int *deviceArray);
//forward declaration
void RunTest()
{
    int* hostArray;
    int* deviceArray;
    const int arrayLength = 16;
    const unsigned int memSize = sizeof(int) * arrayLength;
    hostArray = (int*)malloc(memSize);
    cudaMalloc((void**) &deviceArray, memSize);
    std::cout << "Before device\n";
    for(int i=0;i<arrayLength;i++)
    {
        hostArray[i] = i+1;
        std::cout << hostArray[i] << "\n";
    }
    std::cout << "\n";
}
```

```

cudaMemcpy(deviceArray, hostArray, memSize, cudaMemcpyHostToDevice);
TestDevice <<<4, 4 >>>(deviceArray);
cudaMemcpy(hostArray, deviceArray, memSize, cudaMemcpyDeviceToHost);
std::cout << "After device\n";
for(int i=0;i<arrayLength;i++)
{
std::cout << hostArray[i] << "\n";
}
cudaFree(deviceArray);
free(hostArray);
std::cout << "Done!!\n";
}

```

3.6.3 MyKernel.cu

```

#ifndef _MY_KERNEL_
#define _MY_KERNEL_

__global__ void TestDevice(int *deviceArray)
{
int idx = blockIdx.x*blockDim.x + threadIdx.x;
deviceArray[idx] = deviceArray[idx]*deviceArray[idx];
}

#endif

```


Chapter 4

Data Encryption Standard

4.1 Introduction

Security is a prevalent concern in information and data systems of all types. Historically, military and national security issues drove the need for secure communications. Recently, security issues have pervaded the business and private sectors. E-commerce has driven the need for secure internet communications. Many businesses have firewalls to protect internal corporate information from competitors. In the private sector, personal privacy is a growing concern. Products are available to scramble both e-mail and telephone communications. One means of providing security in communications is through encryption. By encryption, data is transformed in a way that it is rendered unrecognizable. Only by decryption can this data be recovered. Ostensibly, the process of decryption can only be performed correctly by the intended recipient(s). The validity of this assertion determines the “strength” or “security” of the encryption scheme.[15] Many communications products incorporate encryption as a feature to provide security. This application report studies the implementation of one of the most historically famous and widely implemented encryption algorithms, the Data Encryption Standard (DES).[3][5]

- **What is DES?**

Developed in 1974 by IBM in cooperation with the National Securities Agency (NSA), DES has been the worldwide encryption standard for more than 20 years. For these 20 years it has held up against cryptanalysis remarkably well and is still secure against all but possibly the most powerful of adversaries. Because of its prevalence throughout the encryption market, DES is an excellent interoperability standard between different encryption equipment.[16] The predominant weakness of DES is its 56-bit key which, more than sufficient for the time period in which it was developed, has become insufficient to protect against brute-force attack by modern computers. As a result of the need for a greater encryption strength, DES evolved into triple-DES. Triple-DES encrypts using three 56-bit keys, for an encryption strength equivalent to a 168-bit key. This implementation, however, requires three times as many rounds for encryption and decryption and highlights a second weakness of DES speed. DES was developed for implementation on hardware, and DES implementations in software are often less efficient than other standards which have been developed with software performance in mind.[1][3]

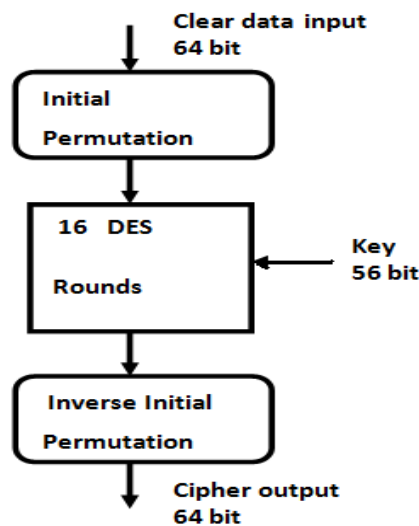


Figure 4.1: Main parts of DES Algorithm

4.2 Modes of Operation for Encryption Algorithms

- a. Block Cipher
- b. Stream Cipher

4.2.1 Block Cipher

An encryption scheme that “the clear text is broken up into blocks of fixed length, and encrypted one block at a time”. Usually, a block cipher encrypts a block of clear text into a block of cipher text of the same length. In this case, a block cipher (figure 4.2) can be viewed as a simple substitute cipher with character size equal to the block size.

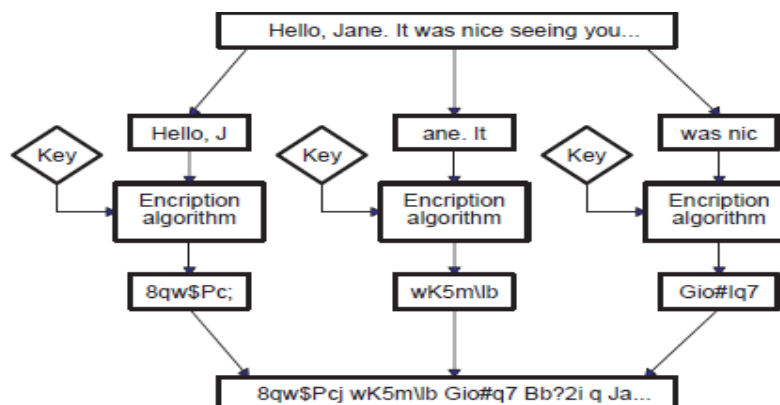


Figure 4.2: Block Cipher

4.2.2 Stream Cipher

Stream ciphers (figure 4.3) operate on streams of data one bit at a time as a continuous stream. DES belongs to a category of ciphers called block ciphers. Block ciphers, as opposed to stream ciphers, encrypt messages by separating them into blocks and encrypting each block separately. Stream ciphers, on the other hand, operate on streams of data one bit at a time as a continuous stream. DES encrypts 64-bit blocks

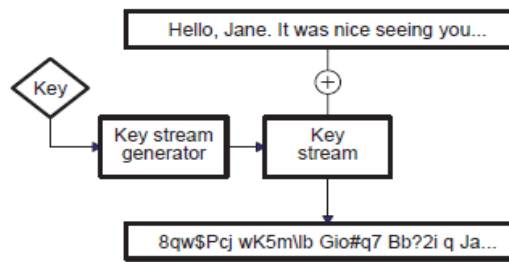


Figure 4.3: Stream Cipher

of plaintext into 64-bit blocks of ciphertext. Plaintext, used in the context of cryptography, is the name commonly given to the body of a message before it is encrypted, i.e. the unaltered text of the message which is to be sent. Likewise, ciphertext is the name commonly given to the encrypted version of the message body which is meant to be indecipherable to any person who does not have the decryption key.

The simplest implementation of a block cipher is to separate the plaintext into contiguous blocks, encrypt each block into ciphertext blocks, and group these ciphertext blocks together as the ciphertext output. This mode of operation is referred to as Electronic Code Book (ECB) mode (figure 4.4). The distinguishing property of this mode is that identical blocks of plaintext always encrypt to the same ciphertext. This is undesirable in some applications.

It is possible to introduce feedback between blocks by feeding the results of the previous encryption block into the input of the current block. The first block of feedback is a randomly generated block called the initialization vector. When this is done, each ciphertext block is not only dependent on the plaintext block that generated it but on each of the preceeding blocks as well, including the initialization vector. Identical plaintext blocks will now only encrypt to the same ciphertext block if each of the proceeding blocks are identical and the initialization vectors are identical. This mode of operation is referred to as Cipher Block Chaining (CBC) mode (figure 4.5).

Other modes of operation exist as well. Two other common modes, Output Feed Back (OFB) mode and Cipher Feed Back (CFB) modes use a block encryption algorithm

to generate a stream cipher.[12][13]

4.3 Modes of Operation for Block Ciphers

4.3.1 ECB (Electronic Code Book)

This is the regular DES algorithm, exactly as described above. Data is divided into 64-bit blocks and each block is encrypted one at a time. Separate encryptions with different blocks are totally independent of each other. This means that if data is transmitted over a network or phone line, transmission errors will only affect the block containing the error. It also means, however, that the blocks can be rearranged, thus scrambling a file beyond recognition, and this action would go undetected. ECB is the weakest of the various modes because no additional security measures are implemented besides the basic DES algorithm. However, ECB (figure 4.4) is the fastest and easiest to implement, making it the most common mode of DES seen in commercial applications. This is the mode of operation used by Private Encryptor.[14][16]

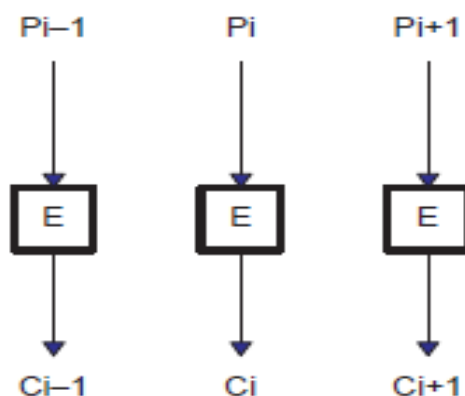


Figure 4.4: Electronic Code Book

4.3.2 CBC (Cipher Block Chaining)

In this mode of operation, each block of ECB encrypted ciphertext is XORed with the next plaintext block to be encrypted, thus making all the blocks dependent on all the previous blocks. This means that in order to find the plaintext of a particular block, you need to know the ciphertext, the key, and the ciphertext for the previous block. The first block to be encrypted has no previous ciphertext, so the plaintext is XORed with a 64-bit number called the Initialization Vector, or IV for short. So if data is transmitted over a network or phone line and there is a transmission error (adding or deleting bits), the error will be carried forward to all subsequent blocks since each block is dependent upon the last. If the bits are just modified in transit (as is the more common case) the error will only affect all of the bits in the changed block, and the corresponding bits in the following block. The error doesn't propagate any further. This mode of operation is more secure than ECB because the extra XOR step adds one more layer to the encryption process. CBC is shown in figure 4.5.

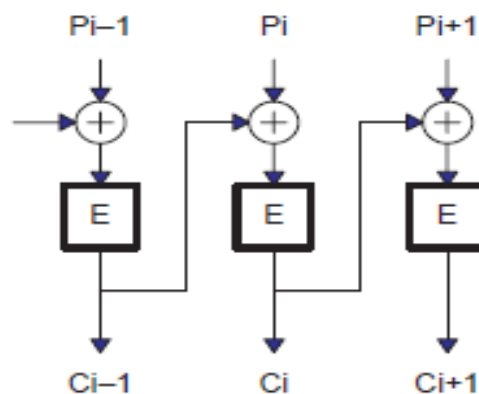


Figure 4.5: Cipher Block Chaining

4.3.3 CFB (Cipher Feedback)

As shown in figure 4.6, blocks of plaintext that are less than 64 bits long can be encrypted. Normally, special processing has to be used to handle files whose size

is not a perfect multiple of 8 bytes, but this mode removes that necessity (Private Encryptor handles this case by adding several dummy bytes to the end of a file before encrypting it). The plaintext itself is not actually passed through the DES algorithm, but merely XORed with an output block from it, in the following manner: A 64-bit block called the Shift Register is used as the input plaintext to DES. This is initially set to some arbitrary value, and encrypted with the DES algorithm. The ciphertext is then passed through an extra component called the M-box, which simply selects the left-most M bits of the ciphertext, where M is the number of bits in the block we wish to encrypt. This value is XORed with the real plaintext, and the output of that is the final ciphertext. Finally, the ciphertext is fed back into the Shift Register, and used as the plaintext seed for the next block to be encrypted. As with CBC mode, an error in one block affects all subsequent blocks during data transmission. This mode of operation is similar to CBC and is very secure, but it is slower than ECB due to the added complexity.

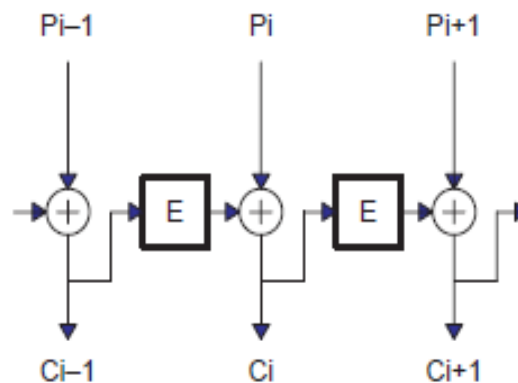


Figure 4.6: Cipher Feedback

4.3.4 OFB (Output Feedback)

This is similar to CFB mode, except that the ciphertext output of DES is fed back into the Shift Register, rather than the actual final ciphertext. The Shift Register is

set to an arbitrary initial value, and passed through the DES algorithm. The output from DES is passed through the M-box and then fed back into the Shift Register to prepare for the next block. This value is then XORed with the real plaintext (which may be less than 64 bits in length, like CFB mode), and the result is the final ciphertext. Note that unlike CFB and CBC, a transmission error in one block will not affect subsequent blocks because once the recipient has the initial Shift Register value, it will continue to generate new Shift Register plaintext inputs without any further data input. However, this mode of operation is less secure than CFB mode because only the real ciphertext and DES ciphertext output is needed to find the plaintext of the most recent block. Knowledge of the key is not required. Figure 4.7 shows output feedback.

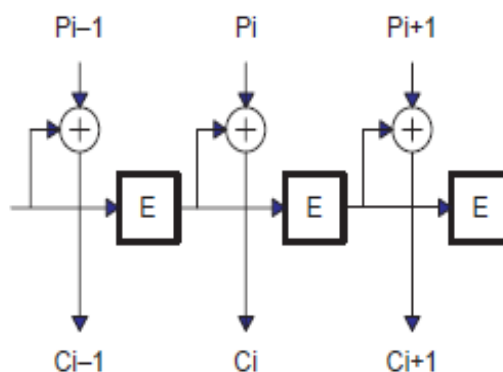


Figure 4.7: Output Feedback

4.4 Conventional DES Implementation

DES relies upon the encryption techniques of:

- confusion and
- diffusion.

Confusion is accomplished through substitution. Specially chosen sections of data are substituted for corresponding sections from the original data. The choice of the

substituted data is based upon the key and the original plaintext.

Diffusion is accomplished through permutation. The data is permuted by rearranging the order of the various sections. These permutations, like the substitutions, are based upon the key and the original plaintext.

The substitutions and permutations are specified by the DES algorithm. Chosen sections of the key and the data are manipulated mathematically and then used as the input to a look-up table. In DES these tables are called the S-boxes and the P-boxes, for the substitution tables and the permutation tables, respectively. In software these look-up tables are realized as arrays and key/data input is used as the index to the array. Usually the S- and P-boxes are combined so that the substitution and following permutation for each round can be done with a single look-up. In order to calculate the inputs to the S- and P-box arrays, portions of the data are XORed with portions of the key. One of the 32-bit halves of the 64-bit data and the 56-bit key are used. Because the key is longer than the data half, the 32-bit data half is sent through an expansion permutation which rearranges its bits, repeating certain bits, to form a 48-bit product. Similarly the 56-bit key undergoes a compression permutation which rearranges its bits, discarding certain bits, to form a 48-bit product. The S- and P-box look-ups and the calculations upon the key and data which generate the inputs to these table look-ups constitute a single round of DES (figure 4.9, 4.10) This same process of S- and P-box substitution and permutation is repeated sixteen times, forming the sixteen rounds of the DES algorithm (figure 4.1, 4.8). There are also initial and final permutations which occur before and after the sixteen rounds. These initial and final permutations exist for historical reasons dealing with implementation on hardware and do not improve the security of the algorithm. For this reason they are sometimes left out of implementations of DES. They are, however, included in this analysis as they are part of the technical definition of DES.[1][12]

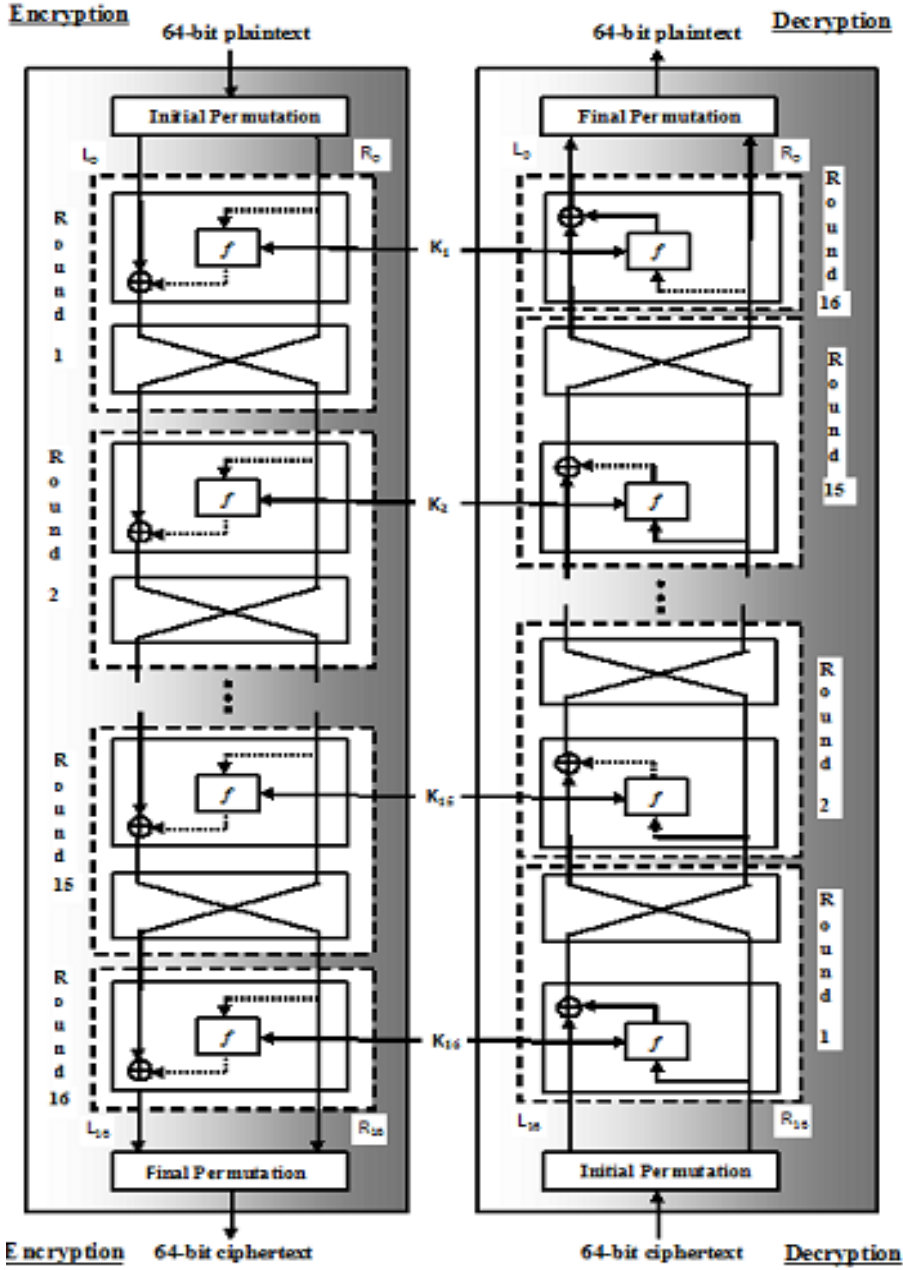
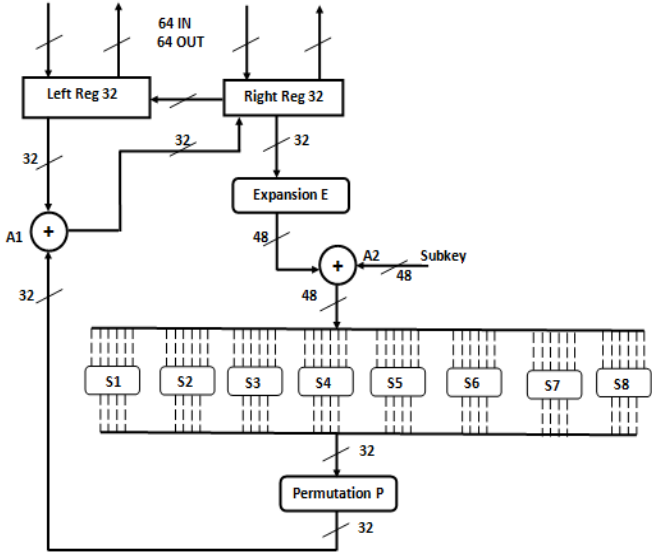
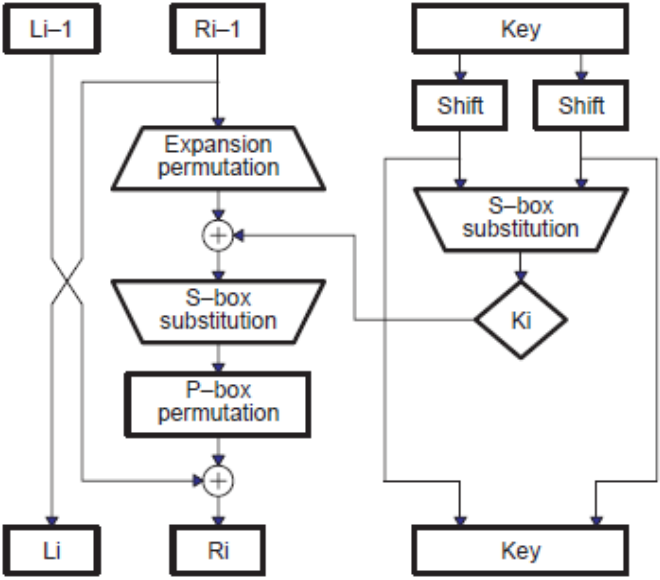


Figure 4.8: DES Architecture



Chapter 5

Implementation

This chapter describes the mathematical model for CUDA-DES implementation along with detailed theoretical analysis. DES is a cryptographic algorithm, making heavy use of bit-wise operations. It encrypts and decrypts data in groups of 64-bits using 64-bits key. For encryption, groups of 64-bits of plaintext are fed into the algorithm to produce groups of 64-bits of ciphertext. The actual experimental results obtained are described in the next chapter.

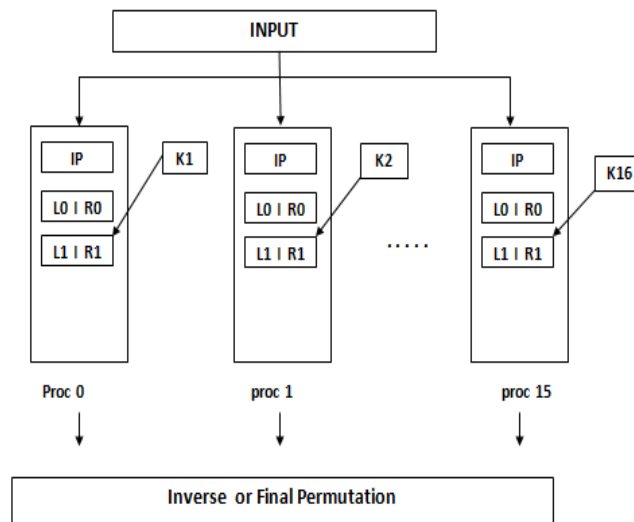


Figure 5.1: Proposed Model

5.1 CUDA-DES algorithm implementation using Proposed Model

Figure 5.1 shows the proposed model for DES encryption. This model takes 64 bits as the input and uses data parallelism approach at the start for parallelization purpose. While the key module uses temporal parallelism. Here we are considering that keys are preprocessed and so readily available during each round of 16 round DES algorithm. So, we are following the standard architecture of DES which is described in chapter 4 along with proposed model for implementation.

5.1.1 Theoretical Analysis for Security Algorithm

Considering combined data and temporal parallelism[17], this method almost halves the time taken by single pipeline.

Let

Number of processes = n ,

Time for one block = p ,

Number of processors = k ,

Time to distribute blocks to k processors = kq .

Observe that this time is proportional to number of processors.

Time to complete n blocks by single processor = np ,

Time to complete n blocks by k processors = $kq + np/k$,

So, Speedup due to parallel processing = $np/[kq + (np/k)] = k/[1 + (k*q/np)]$

If $k*q \ll np$ then speedup is nearly equal to k (i.e. number of processors)

It is found that this will be true if time to distribute the jobs is small.

Now, applying above formula to the proposed model, it is found that speedup is nearly equal to number of processors as the time to distribute the data is very small.

Here in this case, by analysis maximum theoretical speedup found is 6x.

5.1.2 Amdahl's law :

Amdahl's law[16] is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors. The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 as shown in the figure 5.2, no matter how many processors are used. Amdahl's law is

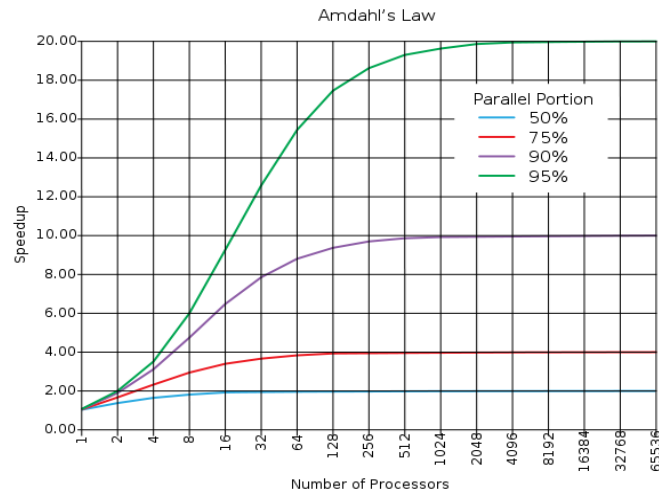


Figure 5.2: Amdahl's Law for Speedup

a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized.

Parallelization :

In case of parallelization, Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the proportion that cannot be parallelized (remains serial), then the maximum speedup

that can be achieved by using N processors is:

$$1/[(1-P)+(P/N)]$$

In the limit, as N tends to infinity, the maximum speedup tends to $1 / (1 - P)$. In practice, performance to price ratio falls rapidly as N is increased once there is even a small component of $(1 - P)$. As an example, if P is 90%, then $(1 - P)$ is 10%, and the problem can be speed up by a maximum of a factor of 10, no matter how large the value of N used.

For example, if for a given problem size a parallelized implementation of an algorithm can run 10% of the algorithm's operations arbitrarily quickly (while the remaining 90% of the operations are not parallelizable), Amdahl's law states that the maximum speedup of the parallelized version is $1/(1 - 0.10) = 1.111$ times as fast as the non-parallelized implementation.

5.1.3 CUDA Program Flow :

CUDA program flow is presented in figure 5.3. This is the important flow which is followed during development of CUDA-DES.

First step is to program start with CUDA event.

We have to create and start timer in next step.

Define number of threads to be used along with memory size. We can test program different number of threads.

After that we have to assign host and device memory.

Setup the execution parameters like block and grid dimensions.

Kernel function is executed in the next step.

Finally stop and destroy the timer.

Free the host and device memory at the end of program.

In our GPU implementation, we implement the data parallel portions of DES by assigning each thread the task of processing one data point. These threads are inde-

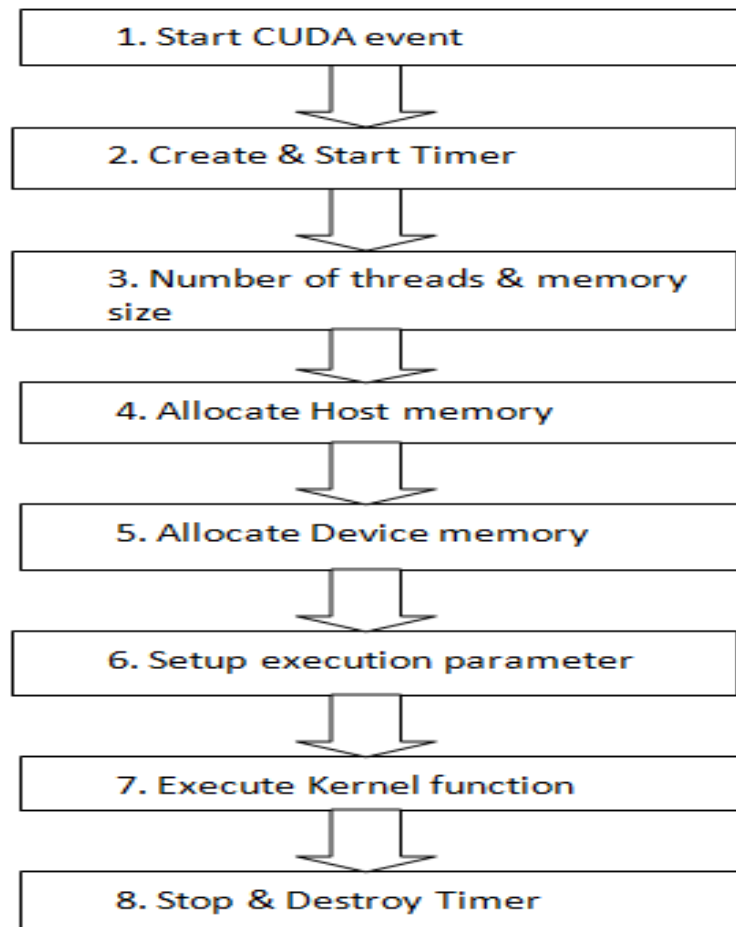


Figure 5.3: CUDA Flow

pendent and can execute in parallel. Datapoints in each permutation of DES can be processed simultaneously.

Parallel Programmability :

- **Parallel Execution :** The powerful compute capability of GPUs stem for their vast available parallelism. In CUDA, we write programs with C-like language. CUDA is currently best suited for a SPMD programming style in which threads execute the same kernel but may communicate and follow divergent paths through that kernel.[18]
- **Synchronization :** Placing barriers in CUDA is somewhat different. CUDA's

runtime library provides programmers with a specific barrier statement, `sync_threads()`, but the limitation of the function is that it can only synchronize all threads within a thread block. To achieve global barrier functionality, the programmer must allow the current kernel to complete and start a new kernel. This is currently fairly expensive, thus rewarding algorithms which keep communication and synchronization localized within thread blocks as long as possible.

Control Flow :

In GPUs, control flow instructions can significantly impact the performance of a program by causing threads of same 32-thread SIMD warp to diverge. Since the different execution paths must be serialized, this increases the total number of instructions executed.

For example, in DES sequential program, the programmer can use it statements to specify permutations, but directly using similar implementation on GPU performs poorly.

Bit-wise Operations :

Bit-wise operations dominate the DES algorithm, which includes the lot of bit permutations and shifting. GPU implementations of these operations become very complex, since they must be built using high level language constructs and fine-grained control flow that tend to cause SIMD divergence.

Our CUDA DES implementation is comprised of several small kernels. Inside a CUDA kernel, we can use fast on-chip shared memory for data sharing, but there is no consistent state in the shared memory, so between two kernels, state flushes are required, then data must be read back for the next pass.[7]

Chapter 6

Result and Analysis

This chapter presents results and analysis for DES and CUDA-DES encryption and decryption. Total three experiments has been performed and obtained results are given in following sections.

6.1 Experiment 1

In first experiment, 64-bit plaintext and key are generated inside the program to produce the ciphertext for CUDA-DES encryption. Initially, encryption has been done for single round of DES on CPU and GPU respectively and then for 16 rounds of DES on CPU and CPU+GPU respectively. Figure 6.1 shows CPU encryption whereas figure 6.2 shows CPU+GPU encryption. The table I shows encryption results.

CPU/GPU	Time (seconds)
CPU	1.185000
CPU+GPU	1.092000

Table I: Results for Encryption


```

Please select from menu:
1. Encryption
2. Decryption
Choice: 1
Enter 64bit text in Hex : 123abc456def7890
Enter 64bit Key in Hex : 1234567890abcdef

des_out = e82826953a1f16e9

It took 0.156000 seconds to complete DES Algorithm using CPU.

```

Figure 6.3: Encryption for 16 round DES on CPU

```

Please select from menu:
1. Encryption
2. Decryption
Choice: 2
Enter 64bit text in Hex : e82826953a1f16e9
Enter 64bit Key in Hex : 1234567890abcdef

des_out = 123abc456def7890

It took 0.156000 seconds to complete DES Algorithm using CPU.

```

Figure 6.4: Decryption for 16 round DES on CPU

6.3 Experiment 3

In this experiment, hexadecimal string file.txt is used as input. Testing is done for speedup using input file sizes like 1KB, 10KB, 100KB. The outputs for CPU and CPU+GPU encryptions for file size 1KB are shown in figures 6.7 and 6.8 respectively. Final results for encryption are shown in table IV. Here it is observed that as file size increases CPU takes more time whereas CPU+GPU takes comparatively less time and so accordingly we get speedup 6.9.

CPU	CPU Encryption	CPU Decryption
Input	123abc456def7890	e82826953a1f16e9
Key	123456789abcdef	123456789abcdef
Output	e82826953a1f16e9	123abc456def7890
Time (seconds)	0.156000	0.156000

Table II: Results for Encryption and Decryption on CPU

CPU+GPU	Encryption	Decryption
Input	123abc456def7890	1e322da2d067052
Key	123456789abcdef	123456789abcdef
Output	1e322da2d067052	123abc456def7890
Time (seconds)	0.125000	0.125000

Table III: Results for Encryption and Decryption on CPU + GPU

	1 KB	10 KB	100 KB
CPU	0.172000	1.435000	7.925000
CPU + GPU	0.156000	1.248000	6.810000
Speedup	1.10	1.15	1.16

Table IV: Results for Encryption in terms of time in seconds



```


C:\Users\Users1\Documents\Visual Studio 2008\Projects\snehal_cuda_des_gpu\Debug\snehal_cuda_des_gpu.exe
Please select from menu:
1. Encryption
2. Decryption
Choice: 1
Enter 64bit text in Hex : 123abc456def7890
Enter 64bit Key in Hex : 1234567890abcdef

des_out = 1e322da2d067052

It took 0.125000 seconds to complete DES Algorithm using CPU + GPU.

```

Figure 6.5: Encryption for 16 round DES on CPU + GPU



```

C:\Users\Users1\Documents\Visual Studio 2008\Projects\snehal_cuda_des_gpu\Debug\snehal_cuda_des_gpu.exe
Please select from menu:
1. Encryption
2. Decryption
Choice: 2
Enter 64bit text in Hex : 1e322da2d067052
Enter 64bit Key in Hex : 1234567890abcdef

des_out = 123abc456def7890

It took 0.125000 seconds to complete DES Algorithm using CPU + GPU.

```

Figure 6.6: Decryption for 16 round DES on CPU + GPU

6.4 Result Analysis

- As first experiment uses different block sizes ranging from 128, 256 and 512, thread size as 32 and 64, the time ranging from 0.140000 to 0.125000 is observed. The optimized result is found when the block size is 256 and thread size is 32.
- From the second experiment it is observed that encryption and decryption takes same time.
- From the third experiment best speedup observed for CUDA-DES encryption is 1.16x for input file of size 100KB. It is clear from theoretical analysis that we could get maximum speedup upto 6x.
- Here, experimental results and analysis results for Amdahl's law (described in

```

Please select from menu:
1. Encryption
2. Decryption
Choice: 1
Enter 64bit Key in Hex : 1234567890abcdef

File Input string 1: 1234567890abcdef
Encrypted Output 1: 516ae556c8f4c96f
File Input string 2: a234567890abcdef
Encrypted Output 2: 619ad4c275ac57f1
File Input string 3: b234567890abcdef
Encrypted Output 3: 67fd13f9bf4c671a
File Input string 4: c234567890abcdef
Encrypted Output 4: ab8659e9be51a619
File Input string 5: 1234567890abcdef
Encrypted Output 5: 516ae556c8f4c96f
File Input string 6: a234567890abcdef
Encrypted Output 6: 619ad4c275ac57f1
File Input string 7: b234567890abcdef
Encrypted Output 7: 67fd13f9bf4c671a
File Input string 8: c234567890abcdef
Encrypted Output 8: ab8659e9be51a619
File Input string 9: 1234567890abcdef
Encrypted Output 9: 516ae556c8f4c96f
File Input string 10: a234567890abcdef
Encrypted Output 10: 619ad4c275ac57f1

It took 0.172000 seconds to complete DES Algorithm.

```

Figure 6.7: CPU Encryption using file

chapter 5) are found to be nearly same.

- So, GeForce G105M gives speedup of 1.16x for the security algorithm using CUDA.

```
Please select from menu:
1. Encryption
2. Decryption
Choice: 1
Enter 64bit Key in Hex : 111

File Input string 1: 1234567890abcdef
Encrypted Output 1: 516ae556c8f4c96f
File Input string 2: a234567890abcdef
Encrypted Output 2: 619ad4c275ac57f1
File Input string 3: b234567890abcdef
Encrypted Output 3: 67fd13f9bf4c671a
File Input string 4: c234567890abcdef
Encrypted Output 4: ab8659e9be51a619
File Input string 5: 1234567890abcdef
Encrypted Output 5: 516ae556c8f4c96f
File Input string 6: a234567890abcdef
Encrypted Output 6: 619ad4c275ac57f1
File Input string 7: b234567890abcdef
Encrypted Output 7: 67fd13f9bf4c671a
File Input string 8: c234567890abcdef
Encrypted Output 8: ab8659e9be51a619
File Input string 9: 1234567890abcdef
Encrypted Output 9: 516ae556c8f4c96f
File Input string 10: a234567890abcdef
Encrypted Output 10: 619ad4c275ac57f1

It took 0.156000 seconds to complete DES Algorithm.
```

Figure 6.8: CPU + GPU Encryption using file

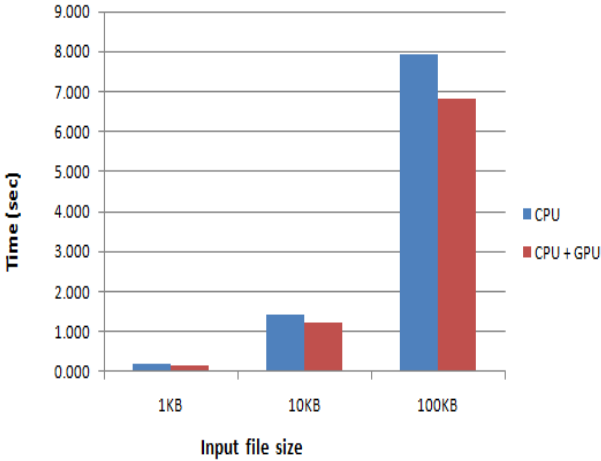


Figure 6.9: Results for Encryption in terms of time

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Implementation of mapping a set of algorithms onto a multi core platform using a CUDA programming model, which describes and controls the communication, concurrences, and synchronization of all components involved is a challenging task.

Considering today's hardware performance, in order to obtain best results, a proper programming strategy for optimum mapping of all processes to existing resources is necessary.

DES is a complex algorithm that uses many effective encryption techniques including a lot of bit permutations and shifting. GPU implementations of these operations become very complex. Hence for DES parallelization experiment, it is observed that GeForce G105M gives speedup of 1.16x for key generation part. From theoretical analysis, it is clear that maximum theoretical speedup that can be obtained is 6x. Hence, GPU can be efficiently used for the parallel program developement.

7.2 Future Work

- There is a scope for parallelization of triple DES and AES as they are more secured.
- To achieve more parallelism in the program for utilizing the high no of cores to achieve good better speedup.
- Also memory as well as code optimization can be useful for getting the better speedup.
- One can use different GPUs to find out the speedup for comparison.

Web References

- [1] <http://www.hbeongpgpu.com/whatiscuda.htm>
- [2] http://www.nvidia.com/object/cuda_get.html
- [3] http://developer.nvidia.com/object/cuda_3_1_downloads.html
- [4] <http://driverscollection.com/?H=GeForce%20G%20105M\&By=NVidia\&SS=Windows%20Vista>
- [5] <http://www.nvidia.com/Download/PreScan.aspx?lang=en-us>
- [6] http://www.nvidia.com/object/cuda_learn_products.html
- [7] <http://www.softpedia.com/progDownload/CUDA-SDK-Download-107212.html>
- [8] <http://www.stillhq.com/gpg/source-1.0.3/cipher/des.html>
- [9] <http://www.eventid.net/docs/desexample.asp>
- [10] <http://impact.crhc.illinois.edu/ftp/report/impact-08-01-mcuda.pdf>
- [11] http://gpgpu.org/wp/wp-content/uploads/2009/06/02-CUDA_basic.pdf
- [12] <http://www.many-core.group.cam.ac.uk/projects/>
- [13] http://www.nvidia.com/object/gpu_technology_conference.html
- [14] http://www.conxx.net/rijndael_anim_conxx.html
- [15] <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>
- [16] <http://en.wikipedia.org/wiki/Amdahl's Law>

Bibliography

- [1] Smid, M.E.,Branstad, D.K.,NBS, “*Data Encryption Standard: past and future*”,Gaithersburg, MD;IEEE,August 2002.
- [2] *Introducing multithreaded programming:POSIX Threads and NVIDIA's CUDA*.
- [3] Seung-Jo Han, Heang-Soo Oh, “*The improved data encryption standard (DES) algorithm*”, Jongan Park,IEEE,August 2002.
- [4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan,Jeremy W. Sheaffer, Kevin Skadron “*A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA*”.
- [5] *Federal Information Processing Standards Publication 46-3,1999 October 25*.
- [6] Greg Ippolito,*YoLinux Tutorial: POSIX thread (pthread) libraries*,Jan 2007.
- [7] Giovanni Agosta , Alessandro Barenghi ,Fabrizio De Santis , and Gerardo Pelosi, “*Record Setting Software Implementation of DES Using CUDA*”.
- [8] CudaReferenceManual.pdf.
- [9] NVIDIA CUDA C Programming Guide.
- [10] David Kirk/NVIDIA and Wen-mei W. Hwu,”Lecture Series” ,University of Illinois, Urbana-Champaign,2007-2009 ECE 498AL.
- [11] Michael D. McCool, “*Scalable Programming Model For Massively Multicore Processors*”,Canada; Proceedings of IEEE,Vol. 96,No. 5,May 2008.
- [12] William M. Daley,Raymond G. Kammer, “*DATA ENCRYPTION STANDARD (DES)*”,FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION,FIPS PUB 46-3,October 1999.
- [13] Brandon P. Luken, Ming Ouyang, and Ahmed H. Desoky, “*AES and DES Encryption with GPU*”,Louisville,August 2002.
- [14] R. Stephen Preissig, “*Data Encryption Standard (DES) Implementation on the TMS320C6000*”,SPRA702,November 2000.

- [15] Behrouz A. Forouzan, Debdeep Mukhopadhyay, “*Cryptography and Network Security*”, 2nd edition, Tata McGraw Hill, 2008, ISBN-0-07-070208-X.
- [16] Andrew S. Tanenbaum, “*Computer Networks*”, 3rd edition, Prentice Hall, 1997, ISBN-81-203-1165-5.
- [17] V. Rajaraman, C. Siva Ram Murthy, “*Parallel Computers Architecture and Programming*”, Prentice Hall, 2000, ISBN-81-203-1621-5.
- [18] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron. and John Lach, “*Accelerating Compute-Intensive Applications with GPUs and FPGAs*”.

Index

Abstract, iv

Acknowledgements, v

Certificate, iii

Conclusion and Future Work, 52

Data Encryption Standard, 28

Implementation, 39

Introduction, 1

Literature Survey, 5

Result and Analysis, 45

Tools and Techniques, 18