ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Βελτίωση απόδοσης μεθόδων κρυπτογράφησης με αξιοποίηση μονάδων GPUs

Διπλωματική Εργασία

Κλαδούχος Δημήτριος

Επιβλέπουσα: Τσομπανοπούλου Παναγιώτα

Βόλος 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Exploiting GPUs for enhanced cryptographic performance

Diploma Thesis

Kladouchos Dimitrios

Supervisor: Tsompanopoulou Panagiota

Volos 2021

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπουσα	**Τσομπανοπούλου Παναγιώτα** Αναπληρώτρια Καθηγήτρια, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Αντωνόπουλος Χρήστος** Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Φεύγας Αθανάσιος** Ε.ΔΙ.Π., Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Ημερομηνία έγκρισης: 17-09-2021

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά την αναπληρώτρια καθηγήτρια κα. Τσομπανοπούλου Παναγιώτα για την εξαιρετική συνεργασία και την συμβολή της στην εκπόνηση της παρούσας διπλωματικής εργασίας καθώς και τους κυρίους Αντωνόπουλο Χρήστο και Φεύγα Αθανάσιο για την συμμετοχή τους στην επιτροπή εξέτασης. Ευχαριστώ, φυσικά, την οικογένειά μου για την πολύτιμη στήριξη που μου παρείχε όλα αυτά τα χρόνια των σπουδών μου ενθαρρύνοντάς με πάντα και βοηθώντας με να τις συνεχίζω απρόσκοπτα. Τέλος, θα πρέπει να ευχαριστήσω τους φίλους μου που βρίσκονταν πάντα δίπλα μου, με στήριζαν και βαδίσαμε μαζί σε αυτό το μεγάλο μονοπάτι στο οποίο αφήνουμε ευτυχισμένες στιγμές και όμορφες εμπειρίες.

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο Δηλών

Κλαδούχος Δημήτριος

17/09/2021

ΠΕΡΙΛΗΨΗ

Μετά από χρόνια έρευνας, οι κάρτες γραφικών (GPUs) βρέθηκε ότι θα μπορούσαν να είναι χρήσιμες σε μεγάλους επιστημονικούς υπολογισμούς, εκτός από την επεξεργασία γραφικών, λόγω του εξαιρετικά φιλικού για παράλληλη εκτέλεση, πολυπύρηνου σχεδιασμού τους. Ως αποτέλεσμα, πλέον χρησιμοποιούνται όλο και περισσότερο στην επιστήμη των υπολογιστών για πολύπλοκους υπολογισμούς και διαδικασίες γενικότερα έναντι των Κεντρικών Μονάδων Επεξεργασίας (CPUs) εξαιτίας της υπεροχής τους σε επεξεργαστική ισχύ. Ένα από τα σημαντικότερα πεδία που οι GPUs μπορούν να είναι εξαιρετικά χρήσιμες είναι η ασφάλεια των υπολογιστικών συστημάτων και η κρυπτογραφία. Όταν εφαρμόζεται ένας αλγόριθμος κρυπτογράφησης, εκτός από το να είναι αποτελεσματικός όσον αφορά την ασφάλεια, η απόδοση της εφαρμογής του είναι κάτι που απασχολεί επίσης. Στην παρούσα διπλωματική εργασία, παρουσιάζουμε μερικές ενδιαφέρουσες μελέτες πάνω στην βελτίωση της αποδοτικότητας που οι σύγχρονες κάρτες γραφικών μπορούν να παρέχουν σε πολύ σημαντικές κρυπτογραφικές εφαρμογές. Δίνουμε έμφαση στην “παράλληλη” εφαρμογή ευρέως χρησιμοποιούμενων συμμετρικών αλγορίθμων κρυπτογράφησης και του αλγόριθμου δημοσίου κλειδιού RSA προγραμματισμένων σε NVIDIA CUDA παρουσιάζοντας μερικές από τις πιο διακεκριμένες έρευνες που έχουν γίνει σε αυτό το πεδίο μελέτης. Αυτόν τον καιρό, η υπάρχουσα βιβλιογραφία γίνεται περισσότερο πλήρης και πολλές ενδιαφέρουσες τεχνικές για βελτίωση της επίδοσης πολλών αλγορίθμων προτείνονται. Τα πειράματα που παρουσιάζονται οδηγούν στο συμπέρασμα πως υπάρχει σημαντική βελτίωση στην απόδοση αυτών των αλγορίθμων όταν επιστρατεύεται η εφαρμογή τους με χρήση GPUs. Εκτός από την παρουσίασή τους, σχολιάζονται επίσης τα αποτελέσματά τους λεπτομερώς, εξετάζοντας επίσης και το περιθώριο για περαιτέρω βελτιστοποιήσεις σε κάποιες περιπτώσεις.

Λέξεις-κλειδιά:

GPU, κρυπτογραφία, συμμετρικοί αλγόριθμοι κρυπτογράφησης, αλγόριθμοι δημόσιου κλειδιού, RSA, απόδοση, αποδοτικότητα, παράλληλη.

ABSTRACT

After years of research, Graphic Processing Units (GPUs) were found that could be useful for heavy scientific computations except for graphics processing due to their highly parallel-friendly multi-core design. Nowadays, as a result they are increasingly used in computer science for complex computations and procedures in general over Central Processing Units (CPUs) because of this superiority in processing power. One of the most important fields that GPUs can be greatly useful is the security of computer systems and especially cryptography. When a cipher is implemented except for ensuring security, performance is an aspect that matters too. In this diploma thesis, we present some interesting studies on the ciphers efficiency improvement that modern GPUs can provide for critical cryptographic applications. We focus on parallel implementation of widely used symmetric-key ciphers and the RSA public-key algorithm programmed in NVIDIA CUDA by presenting some of the most distinguished research that have been made in that field of study. These days, the existing literature is becoming more complete and many interesting techniques for enhancing the performance of the existing ciphers are being proposed. The experiments presented result in that there is a significant improvement in the performance of these cryptographic applications when GPUs implementations are deployed. Except for the presentation of them, these results are finally discussed in detail, examining the room for further optimizations too in some cases.

Keywords:

GPU, cryptography, symmetric-key ciphers, public-key ciphers, RSA, performance, efficiency, parallel.

TABLE OF CONTENTS

List of figures

List of tables

CHAPTER 1

INTRODUCTION

Central Processing Units (CPUs) had been for a long time the only units available for general purpose computing and as a result for large-scale computing too. But along with the evolution of technology, the “large-scale” meaning changed significantly over the years and new complex applications with higher computational requirements appeared. That need led to research for new effective ways to meet those requirements. The best answer was given by Nvidia in 2007 which introduced a new method which played a major role in accomplishing what we now call High Performance Computing (HPC). That was the exploitation of Graphic Processing Units (GPUs) for general purpose computing (GPGPUs) which was achieved in practice with the use of a new programming language named CUDA (Compute Unified Device Architecture) used for programming the Nvidia GPUs. That is why modern programmable GPUs are also called General Purpose ones (GPGPUs). Thereafter, few more similar functionality languages were created (OpenCL, OpenACC, C++ AMP, etc.), but still CUDA remains the most commonly used for professional high performance computing around the world.

GPUs have many important advantages in computing ability over the CPUs. First of all, GPUs are ideal for parallel processing due to the hundreds of cores they consist of in comparison with CPUs which have only a few. GPU cores can handle thousands of threads simultaneously while CPU cores can use a small number of threads per core. Additionally, CPUs may have low latency but for server environments cluster computing used requires high hardware investment and consumes a lot of energy. On the contrary, GPUs have lower power consumption and in terms of performance provide much higher throughput and bandwidth which is more critical in achieving efficient computing. In general, they significantly improve the responsiveness and the speed of a wide spectrum of applications in a large number of market fields such as scientific software, visual processing, neural networks’ implementation and a range of some critical security applications as of late.

Cryptography is one of these fields, providing confidentiality, privacy preservation, integrity and authenticity to the users, but it is often time consuming to implement them. The present diploma thesis is an overview of some remarkable optimization attempts in the most used cryptographic algorithms that have been made so far. Some important ciphers included among others are AES,DES,RSA etc. All optimizations were implemented using Nvidia’s CUDA programming language. The ciphers included were parallelized in the most effective way possible taking advantage of the full functionality of the GPGPUs. The results show a clear and significant enhancement to the performance of the implemented ciphers.

The rest of the diploma thesis is organized as follows. Chapter 2 is a roundup of the functionality and programming capabilities of GPGPUs. Chapters 3 and 4 present the optimization methods that have been attempted so far in symmetric-key block ciphers and the most used public-key cipher, the RSA respectively. Chapter 5 presents our conclusions and discusses future work.

CHAPTER 2

THE GPGPU PROGRAMMING MODEL

2.1 NVIDIA GPGPU parallel architecture

The modern GPUs[1] are characterized by their high bandwidth and high throughput, computing power and energy-efficient functionality. Their key-feature is the massive multithreading ability which along with the shared control logic across the threads succeeds to hide the latency during processing.

The most widely used GPUs for general purposes are the Nvidia GPUs. Before analyzing their architecture, we need to declare the terms we are going to use henceforth. A program running on the CPU is the *host* program*,* GPU is the *device* and the code which is going to be executed in the GPU (device code) is called through the *kernel* launch which acts as a function that runs on the *device.* The whole process of a CUDA program specifically starts with a host program with one or multiple threads running on a CPU which consists of one or more executable parallel “kernels” ready to be launched from the suitable host code point.

Nvidia GPGPUs are designed to support a large number of threads which are able to run simultaneously. But kernels are organized in a specific way and have strict functionality rules. Before launching a kernel, its size has to be determined. For that reason, the number of grids, thread blocks and each block’s number of threads needs to be set. The maximum numbers of these values may differ per GPU according to their specifications. As for these terms, thread blocks are a programming abstraction which depicts a group of threads being able to be executed serially or in parallel. A grid is formed by the grouping of multiple thread blocks. The computing architecture is configured with a group of streaming multiprocessors (SM) containing 64-128 cores each in most cases, depending on the architecture. The microarchitecture of the GPU defines the organization of some elements inside the SMs which schedule, transfer and execute. The occupancy of a GPU mainly defines its computing capability and of course different architectures to enhance performance are being tested and applied over the years. In order to comprehend the computing capability, we mention some critical characteristics of one state of the art Nvidia GPU. Ampere architecture, is one of the newest Nvidia GPU architectures and the documentation which Nvidia provides for the A100 graphics card which is based on this architecture, details its characteristics as following: 108 SMs, 64 FP32 cores/SM, 1024 maximum threads/block, 2048 maximum threads/SM, 32 maximum thread blocks/SM and 164 KB shared memory size/SM. In short, the whole procedure for the processing through a GPU starts with the work scheduler which distributes CUDA thread blocks to SMs, the multithreaded SMs schedule and execute the CUDA thread blocks and individual threads which are assigned to them and then each SM can process multiple concurrent threads hiding some long-latency loads from DRAM. After a thread block executes its kernel code, its binded SM resources are released so the work scheduler can assign a new thread block to those SM.

2.2 GPU memory structure

It is important to declare that each thread block is assigned to and executed on a single SM. As regards the CUDA supported memory organization of a GPU, the hardware implementation of memories is the following:

- **Per thread memories**
  - **Local memory**, where registers and other thread data are stored when there are no more SM resources or when it is chosen for a reason. This is also the memory which array type variables are stored in. It is a memory that provides slow-speed data access and can be even 150 times slower than registers or shared memory, which we subsequently mention. The lifetime of the local memory is the same as that of the thread's lifetime. (Read/Write)
  - **Registers,** which are used for stack variables which are declared in kernels. Each SM consists of thousands of registers. It is the fastest form of memory with a lifetime the same as that of the thread's lifetime. (Read/Write)
- **Per thread block memory**
  - **Shared memory**, which is an L1 data cache in which all threads of a thread block have access, so we have an L1 cache (up to 164KB in Nvidia A100/Ampere architecture) per SM. So being located inside each SM makes

them have very low latency. It is a fast-access memory with a lifetime same as that of the thread block's lifetime. (Read/Write)

- **Per grid memories**
  - **Global memory**, in which all the essential variables needed from the host code are transferred to at first place and all of the variables that contain are accessible from all the thread blocks of the grid. For uncoalesced reads and writes creates huge application bottlenecks and in general is the slowest access memory. Its lifetime lasts until the CUDA application is terminated. Its capacity is about 40GB (Nvidia A100/Ampere architecture) (Read/Write)
  - **Constant memory**, in which all the thread blocks of the grid have access. It is useful for constants that cannot be compiled into the program. It is a high-speed data access memory with the lifetime of the application which participates in. (Read/Only)
  - **Texture memory** is a type of cache memory rarely used for general purpose computing so it is not of primary importance discussing their functionality. (Read/Only)
- **Combined global and local memory**
  - **Level 2 cache** shared by all the SMs (up to 1MB).[1]

2.3 Kernel execution process and attributes

The basic units of SM managing are called warps, which are responsible for scheduling and executing threads. Threads in the same warp start at the same time at the same program address and are executed simultaneously with only one warp being executed each time. However, when designing a CUDA code we need to have in mind that in fact 16 threads (a half-warp) are executed simultaneously because each 32 threads' same single instruction needs two clock cycles to be executed and warp scheduler to issue the next instruction due to the fact that many instructions are moving through the pipeline at once. In CUDA the sequence of operations that execute on the device in the order in which they are issued by the host code is called a stream. CUDA gives programmers the opportunity to interleave different operations from different streams or when possible run concurrently.

In order to use the asynchronous stream functionality which mentioned above, a memory mechanism for the host memory is necessary to be used. This is the page-lock memory (or pinned memory) and has the capability to concurrently execute kernels and memory transfers eliminating the need for allocating device memory thread blocks. GPUs get the data they need from the CPU using the Direct Memory Access mechanism (DMA). It is an effective manner to transfer data to or from the RAM (where the pinned memory is stored to) without involving the CPU in the whole process and of course not generating any page fault on the accesses. The other memory design memory characteristic which is very crucial for effective performance in CUDA programs is the bank conflicts which may be caused during shared memory access. First of all, let us define how the shared memory is organized. Supposing the memory has the form of a matrix, each column is a bank and in Nvidia GPUs there are 16 or 32 banks, in which memory locations are stripped across them in units of 32 bits. If the threads of a half warp happen to access data that belong to the same bank we call this phenomenon a bank conflict and has as a result a significant drop in performance especially if it happens repeatedly. So it is very important for the programmer to be careful to access different banks in every half-warp execution[1].

CHAPTER 3

SYMMETRIC-KEY BLOCK CIPHERS

3.1 The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a widely used and one of the most important ciphers and is used worldwide in many critical fields. Developed by the Belgians Vincent Rijmen and Joan Daemen (hence its original name, Rijndael), is an electronic data cipher created by the U.S. National Institute of Standards and Technology (NIST) in 2001. It is the first publicly accessible cipher approved by the U.S. National Security Agency (NSA) which uses it for greatly confidential information.

AES is a block cipher[3], meaning that the algorithm operates on a fixed-length group of bits which in this case are 128 and can be used with three different available key versions: a 128-bit sized, a 192-bit and a 256-bit with the internal rounds of the cipher being 10, 12 and 14 in each case respectively. AES encrypts all 128 block bits in one iteration and that is one reason for the comparably small number of rounds. In general, the algorithm consists of the following steps, called layers with each layer manipulating all of the 128 data bits:

1. **Key Addition Layer** A 128-bit key (subkey) which is derived from the original key through the key scheduler is being XORed to the data path. (The first round consists only of that layer).
2. **Byte Substitution Layer (S-Box)** The data path's elements are non-linearly transformed through lookup tables using a method based on the mathematical properties of Galois Fields. The contribution of this layer to the whole procedure is that it achieves confusion to the data which means that all changes in each state are spread quickly throughout the data path.
3. **Diffusion Layer** It contains two sub-layers: the shiftRows layer and the MixColumn one. This layer is all about the diffusion over the bits being processed. The two sublayers perform linear operations and are the following:
    a. **ShiftRows Layer** This sublayer is responsible for byte-level data permutation.

b. **MixColumn Layer** This sublayer executes matrix operations which mix blocks of four bytes (last round does not make use of it which makes the whole scheme symmetric).

In the decryption process, since AES does not follow the Feistel structure but still a symmetric cipher, there must be an inversion in all layers separately which turn out to be similar to the encryption ones.

As for the implementation of the AES and other block ciphers like DES which will be discussed later on, there is not a unique way to achieve it but five different modes of operation. These are the following:

- **Electronic Code Book mode (ECB)**, in which if, for example, the block cipher encrypts and decrypts in blocks of *x* bits and the total message size is not a multiple of *x* bits, it must be padded so as to be before the encryption process. Subsequently, the algorithm is applied independently to each block. The main disadvantage of this mode is that it is highly deterministic i.e. in the case that the same key is used, the encrypting identical blocks results in identical ciphertext blocks which does not offer the desirable security levels.
- **Cipher Block Chaining mode (CBC)**, in which every ciphertext block does not depend only on its own plaintext but on all the previous ones. The result of the encryption of every block is *XORed* with the succeeding plaintext block and the sum is then encrypted producing the next ciphertext block. An important attribute of this mode is that the procedure becomes random with the help of an initialization vector (IV) in the beginning of the implementation and is different in every encryption process in order to avoid repetition.
- **Cipher Feedback mode (CFB)**, which is similar to CBC mode. Assume an x_0 with the role of the IV as used in CBC and with a size of *n* bits. At first it is being encrypted and a 128-bit ciphertext is created. Then, the first *s* bits ($n > s = message\ block\ size$) of the 128 resulted ciphertext bits are *XORed* with the message block's 128 bits and the result of this operation make the *s* right bits of x_1 while the left *n-s* bits of $x_1$are the right *n-s* bits of x_0 and so on.

- **Output Feedback mode (OFB)**, which has almost the exact same procedure to CFB but with the difference that x_i has the result of the *s* ciphertext bits ensued from x_{i-1} block encryption but before being *XORed* with the message block. The first *n-s* bits are ensued from the first *n-s* bits plaintext block of x_{i-1} as well.
- **Counter mode (CTR)**. CTR is another mode that handles the block ciphers as stream ciphers. Every encrypted block is created from a one-time unpredictable value which is appended to a counter number and then is AES encrypted. The complete ciphertext block is consequently *XORed* with the corresponding message block.

3.1.1 CUDA optimization attempts on ECB and CBC mode

An interesting CUDA optimization model for AES cipher was proposed by Li et al.[2]. Their implementation focuses specifically on the ECB for encryption and CBC mode for decryption which are considered some of the most suitable for parallelization. The 128-bit key version was selected. The encryption scheme is as we described it previously but in this case the researchers implemented it with an important diversification, which is the use of T-Boxes. T-Boxes were proposed by Rijndael designers as an alternative method for faster software implementations [3, p. 115]. The main idea is to combine the functions, but not the key addition one, to create one look-up table. So the result is four tables, each consisting of 256 entries. Each entry consists of 32 bits. These tables are the T-Boxes. The T-box takes one byte as an input, and creates a 32-bit column vector. Each transformation round conducts the following computations:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}] \oplus k_j$$

where *a* depicts the input per round, k_j is a column from the key used in the stage and e_j signifies one column of the round output in relation to the bytes of *a*. The advantage of that method is that only four table accesses are enough to get an output of 32 bits of one round. Hence, only 16 table look-ups are needed to compute one round. The scheme they worked on is the 128-bit sized key. The experiment is substantially divided into two different ones for encryption and decryption processes. The encryption process optimization is attempted on the ECB mode which the researchers deem to be more suitable for parallelization. On the contrary, they try to optimize the CBC mode for decryption. CBC's decryption input blocks are immediately available and as a result the inverse cipher operations used in decryption can be performed in parallel.

As for the memory usage scheme and the granularity of the parallel processing, the implementation of AES comprehends a 16-byte usage for each thread meaning that each thread handles a single 16-byte AES block and all blocks are processed concurrently. Due to the fact that each thread's computation does not require parallel execution, synchronization or data sharing among other threads, they can use only their own registers during processing. So for the ECB encryption, the load of four continuous words should be done (block of 128 bits) into global memory at first but the fact that the access pattern of global memory for a single 32-bit word by all threads included into the half warp is non-sequential and separated creates low bandwidth. A solution to this problem could be the reorder of the plaintext in host memory before it is transferred to the GPU global memory. Firstly, each plaintext block is divided into four 32-bit words and storage be columnar so that different 32-bit words which belong to different blocks are consistent in host linear memory space. By this way, the half warp's 16 threads could read these consistent words from global memory, so the memory coalescence has been succeeded. It is noteworthy that this reorder process could be done efficiently by loading the data for encryption from disk files. The CBC decryption process is very similar to the ECB encryption except from the fact that the previous ciphertext blocks should be *XORed* with the decrypted ones so as to recover the plaintext. In order to avoid loading once again the encrypted blocks from global memory for XORing, the already loaded in registers ciphertext blocks by each thread can be stored into shared memory for much higher efficiency when applying the inverse cipher function. Taking into account that the round key values and the T-boxes are read-only data and need to be shared among all threads, the expected option would be to store both those values into the constant memory. However, because the access of the T-boxes is random (not as the Round Keys), given that their quick access is very important for the whole encryption process and that they require only 4KB of storage they chose to store them in the shared memory to ensure the fast access.

The specifications of the environment are listed in Table 3.1 below. Some early results showed that using 512 threads per block exploits the maximum rate of occupancy of the streaming multiprocessors and provides the highest possible computational speed, so this setting was selected as the default one. They also set the shared memory capacity to 48KB and L1 cache to 16KB and enable the ECC check which automatically fixes possible data errors in RAM.

Table 3.1. Specification of experiment platform

Platform	Inspur NF5588
CPU	Intel(R) Xeon(R) E5620@2.4GHz
Memory	24 GB
OS	Windows 7(64-bit)
Compiler	Visual C++9.0(option –O2)
GPU Accelerator	NVIDIA Tesla C2050
GPU Memory	3GB
PCI Bus	PCI-E 2.0×16
CUDA Compiler	Nvcc Ver4.0

The shared memory which was used in CBC mode decryption from each block is 8KB deriving from the number of threads per block (512) multiplied by the size of the AES block which is thread handles (16 Bytes). The SMs of the Nvidia Tesla C2050 which they used in the experiment, are able to have three thread blocks working concurrently. Both modes in encryption and decryption could make full use of the SMs. The major achievements of the experiment are coalesced memory access whis previously mentioned and the high speed grouping of the low latency registers. The highest throughput achieved is 60GB/s while the theoretical one was 144GB/s and 1.03Tflops computing capability. Analyzing the whole process we find that computing time is not much in AES. The reason for the utilization rate being low is mainly the regular random access to the T-Boxes in shared memory which has as a result a number of bank conflicts to happen. It is noteworthy that this GPU implementation resulted in a 50 times speedup in comparison with the Intel Core i7-920 2.66GHz CPU implementation which reached a 1.2Gbps throughput. As regards the data transfer and processing, at first it is important to hide the time that these procedures consume by adopting some clever techniques. When studying a GPU computing capacity we do not want to include the data transfer time to calculate the throughput but we need to consider and calculate this cost when evaluating the actual promotion of GPU computing. PCI-E bus is responsible for data transfer between host and device memory when the GPU is working. PCI-E 2.0x16 which the researchers used could theoretically provide an 8GB/s throughput but in the experiment was only about 3GB/s with the time of data transferring being more than the time consumed for computing. In the experiment, the data transfer time from CPU to GPU is larger than time of computing. With data transfers taken into account, the maximum throughput observed is not much, being just 11.3Gbps. As a promising optimization attempt, they used the stream mechanism which CUDA provides so as to overlap data transfer along with kernel execution in the decryption process. Table 3.2 shows the different numbers of streams tested in the experiment along with their impact on the algorithm’s performance for varying amounts of input data. The cost of pinned memory allocation time is also depicted. In general and theoretically speaking, the more streams we use the more data transfers and kernel executions are hidden and as a result the total time cost should be decreased. However, in practice Table 3.2 shows that after 4 streams or 8 in some cases, as more streams are used, the time increases. The reason is that except for the stream's usage cost, the speed of data being transferred through PCI-E bus of the overlapping data along with the kernel execution decreases significantly as this is ascertained by the Visual Profiler. The use of the asynchronous stream mechanism requires the usage of page-lock memory and its allocation time cannot be ignored. However, this method is not always a good idea to be used because the experiment showed that in some cases the decryption of larger files with this technique used provides lower throughput than smaller files decrypted with the same technique. The limitation of PCI-E bandwidth makes it impossible for the actual throughput of the GPU implementation of the AES algorithm to be significantly increased with even more powerful GPUs. So we understand that the bus' bandwidth along with the pinned memory cost of allocation are considerable limitations for every optimization attempt and should be taken into consideration in order to find the best possible combination of the CUDA existing optimization techniques.

Table 3.2 Total cost using streams[2]

Data Scales(KB)	0 streams	2 streams	4 streams	8 streams	16 streams	Pined memory allocation time (ms)
256	0.44	0.20	0.19	0.23	0.29	0.30
1024	0.96	0.45	0.43	0.60	0.67	0.31
4096	3.15	2.23	2.10	2.06	2.09	0.93
16384	12.58	6.69	6.12	5.88	7.94	3.36
65536	48.06	35.12	32.83	23.45	31.23	13.12
262144	189.50	140.20	97.03	92.47	124.36	58.5

3.1.2 Testings based on allocation strategies and granularity

Another study about an optimized CUDA AES implementation for maximized throughput was made by Iwai et al.[5], in which they examined the relation between different memory allocation strategies for the AES parameters and the parallel processing granularity (the amount of work/computation assigned and performed by a task). The implementation of the algorithm is the one with the 128 bit size of key and the 10 rounds, is based on an optimized version of ANSI C source code for cipher which is included in the OpenSSL library; the open-source toolkit for SSL/TLS protocol and uses the T-Box method which we described in the previous experiment. The selected mode is the ECB which as we mentioned before is suitable for parallelization. Important implementation details are that the round key was generated by CPU once and after that transferred to the GPUs global memory, the initial text for encryption was made with random value and was generated by CPU too and the plaintext had a fixed size of 256MB in order to measure the peak performance of each implementation. Thread blocks were chosen to be 60 along with a fixed number of threads 512 for all the executions.

A crucial implementation aspect was the computation method of the T-boxes. T-box consists of 256 entries each of them having 32-bit data. The requirements for each round are four T-box transformations which are conducted from the shifting operation applying one T-box. An alternative manner for performing these T-box transformations is by using 4 pre-computed T-boxes but this would be not preferable as it would require four times more memory space than just one T-box. Four different granularity implementations were used to test which of them achieves the best throughput in combination with the memory scheme we will describe. In the first one, each thread was set to handle 16 bytes of data meaning that is assigned to a single plaintext block which as we know has the same size. Since the data which are processed in parallel are independent from each other the advantage of this implementation is that there is no need for synchronization and sharing data between threads. The second attempt was 8 Bytes/thread and 4 Bytes/thread with the former needing to employ two threads to encrypt each block and the latter four. In this case data sharing and synchronization is required. The last method assigned 1 Byte only to a single thread. The AES encoding algorithm using this study is optimized for 32 bit processing so we understand that it is better to process AES with a 32 bit operation unit at least. 1 Byte for a thread is still feasible for the reason that AES has been designed with an 8 bit operation unit and this implementation means that 16 threads are required for each plaintext block. This granularity does not provide a good performance though, because the GPU used has a 32 bit operation unit. The memory usage scheme includes the store of the read-only data of T-boxes and Round Keys on constant memory because we need them to be able to be shared among all threads of the grid. The plaintext storage is the other element which is crucial for the algorithm's throughput maximization. At first place it is stored in the global memory and then is sequentially loaded on shared memory when encoding is started. This does not happen only in the case of the 16 Byte/thread granularity in which the whole plaintext and the intermediate values are allocated on registers. Researchers implemented two different types of memory allocation patterns to determine the most efficient; the Array of Structure (AoS) and the Structure of Array (SoA). AoS handles the plaintext as it is while SoA allocates each element of the plaintext needed into one array. Bank conflict problems emerge in different ways in each case and the reduction of its occurrences requires better allocation pattern selection for each implementation. Figure 3.1 and Figure 3.2 show these access patterns of threads to shared memory in case of the granularity at 8 Bytes/thread.

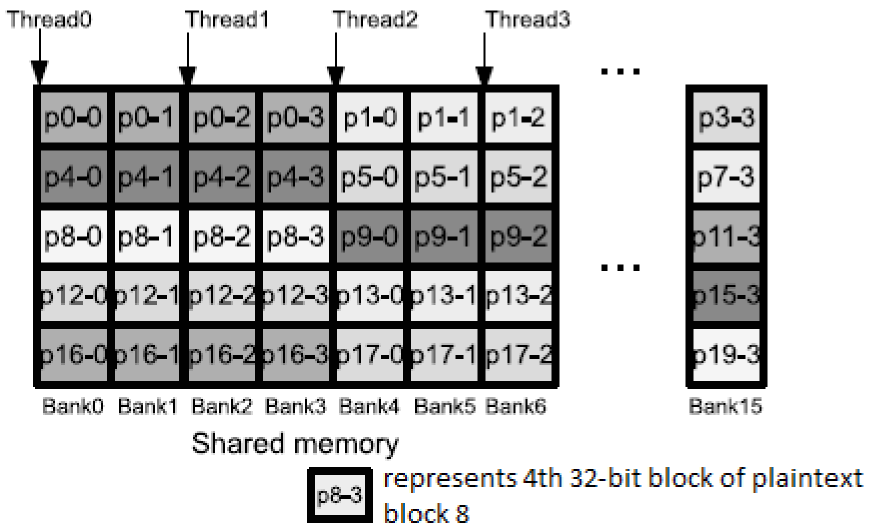

Thread0
Thread1
Thread2
Thread3
p0-0 p0-1 p0-2 p0-3 p1-0 p1-1 p1-2 p3-3
p4-0 p4-1 p4-2 p4-3 p5-0 p5-1 p5-2 p7-3
p8-0 p8-1 p8-2 p8-3 p9-0 p9-1 p9-2 p11-3
p12-0 p12-1 p12-2 p12-3 p13-0 p13-1 p13-2 p15-3
p16-0 p16-1 p16-2 p16-3 p17-0 p17-1 p17-2 p19-3
Bank0 Bank1 Bank2 Bank3 Bank4 Bank5 Bank6 Bank15
Shared memory
p8-3
represents 4th 32-bit block of plaintext block 8

Figure 3.1: Array of Structure, an allocation of plaintext[5]

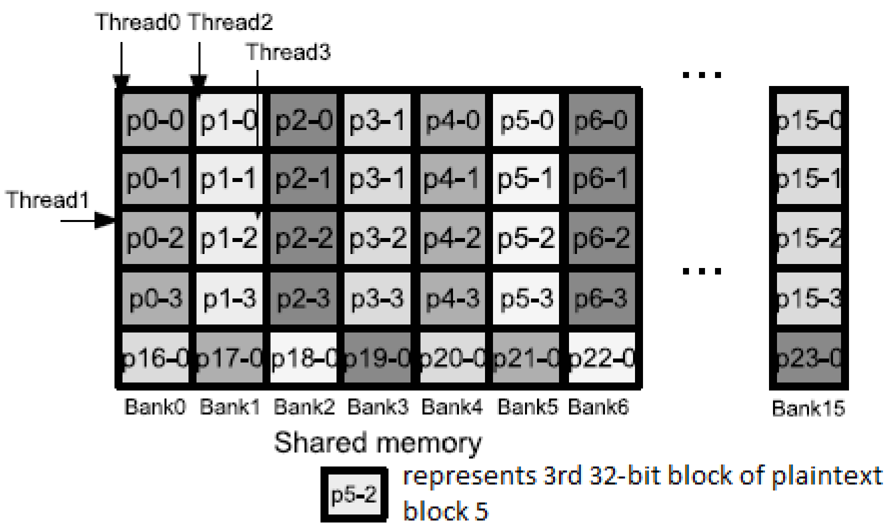


Figure 3.2: Structure of Array, an allocation of plaintext[5]

Thread block switching which causes a significant overhead can be cut down. Simply mapping each thread with a plaintext (as usual with most CUDA applications) is the reason for that. The solution to this problem is reusing the same threads after finishing encrypting each plaintext by returning to the starting point and continuing to encrypt other plaintexts again and with this method we accomplish to encrypt a number of plaintext by using few threads. As for the overhead caused by data transfers it was encountered using a combination of AES encryption process and overlapping transferring plaintext and ciphertext both from and to global memory. Figure 3.3 shows this adopted overlapping technique.

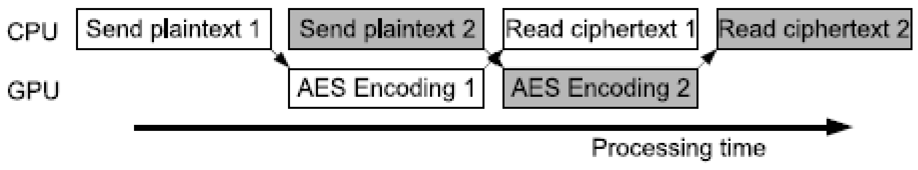


Figure 3.3: Overlapping technique with data transfer and processing[5]

Table 3.3 shows the environment in which the experiment was conducted with the GPU Accelerator being an NVIDIA Geforce GTX 285. Finally, Table 3.4 depicts the results of the experiment and provides a comparison between different implementation combinations. All of them applied four T-boxes. Testings before the experiment indicated that this implementation enhanced the performance compared to one T-box implementation due to not much computation required and eventually the memory space for four T-boxes did not affect the performance. Both Structure of Array and Array of Structure allocation methods were evaluated except for the cases of granularity at 16 Bytes/thread and 1 Byte/thread. In the former granularity case, shared memory to be used for plaintext was not needed because registers could replace the shared memory and the allocation registers for plaintext did speed up the performance which the allocation at shared memory did not because it caused no memory conflicts and also no operations for memory address computation. The results showed that one of the implementations with granularity at 16 Bytes/thread achieved the highest 35.2 Gbps throughput. The reason for that is that in this case no shared memory is required for processing AES encoding meaning that there is very fast register access and no bank conflicts. No synchronization is needed too.

Table 3.3. Specification of experiment computer

CPU	Core i7 Quad i7-920(2.66GHz)
Memory	6 GB
OS	CentOS 5.3 (kernel ver2.6.18)
Compiler	gcc ver 4.1.2 (option -O3)
GPU Accelerator	NVIDIA Geforce GTX 285

GPU Memory	1GB
CUDA Compiler	nvcc ver2.3

As for the allocation place selection of the T-boxes, we can see that constant memory causes a drop in the algorithm's efficiency despite the fact that it gives enhanced access performance with coherent memory access because the constant memory equips the cache system. The disadvantage is that T-box transformation provides random access, but fortunately shared memory can adapt to accessing memory access randomly. So it is obvious that it is preferable to process T-box transformations in shared memory (some implementations with specific granularity did not accept allocating T-boxes on the constant memory). The round key best allocation place decision is a bit more difficult. The allocation of the round keys on the shared memory proved to be approximately 2% faster than the allocation of constant memory. Round key access needed coherent memory access and that is why they almost provided the same throughput. In the other cases synchronizations, access of shared memory and bank memory conflicts played a major role in the drop in performance. As for the 8 Byte/thread and 4 Byte/thread, allocating round keys on constant memory caused less bank memory conflicts than accessing shared memory contrary to the 16 Byte/thread case. The AoS plaintext allocation implementation performed about 1.5 times better performance than SoA implementing 4 Byte/thread granularity. SoA implementation gave twice the bank conflicts than AoS because this one provided four-way bank conflict. AoS did not perform any bank conflict without T-Box transformation. As for the granularity of 8 Bytes/thread, the performance divergence was not significant between SoA and AoS implementation because the memory access pattern between threads was similar on each implementation. Finally 1 Byte/thread granularity achieved very low throughput as expected because the implemented algorithm was created using 8 bit operation which was made from algorithm optimized to 32bit operation divided into four operations.

Table 3.4 Throughput of each implementation[5]

Granularity	16 B/th	16 B/th	16 B/th	8 B/th	8 B/th	8 B/th	4 B/th	4 B/th	4 B/th	1 B/th	1 B/th
T-Box	Constant	Shared	Shared	Shared	Shared	Shared	Shared	Shared	Shared	Shared	Shared
Key	Constant	Constant	Shared	Constant	Shared	Constant	Constant	Shared	Constant	Shared	Constant
Memory Allocation	N/A	N/A	N/A	AoS	AoS	SoA	AoS	AoS	SoA	AoS	AoS
Throughput (Gbps)	5.0	34.4	35.2	26.9	23.9	23.4	25.3	25.0	17.1	2.9	2.6

3.1.3 An application-oriented data structure

An interesting application simulating strategy for block cipher processing on GPU for AES was proposed by Harrison and Waldron[4]. The model was a data structure used for cryptographic representations of client requests. They examined the problems that may come up from the mapping of this structure to the GPU. The main encryption modes of operation were analyzed outlining the performance and behavioural implications when executing them under this model. The implementation of AES is used as an underlying block encryption algorithm to show the overhead caused by moving from an optimized hardcoded approach to a generalised one. The key for a good performance of a client cryptographic application is to be able to map the process elements onto GPUs in a simple way. The proposed data model is designed in a way to encapsulate cryptographic functions suitable for GPU usage. The experiment is focused on how the plaintext data can be mapped to the GPU threading model for both serial and parallel in nature modes of operation. The objective of the research is to determine the overhead in association with the data model and its mapping ability and the accent of it is also the issues emerging from the mixing of modes of operation within only one GPU call. The characteristics of the proposed model are obvious that make it suitable for types of applications working on servers which require massive and continuous data encryption such as high bandwidth media streaming and secure backup/restore.

The optimized implementation of AES is in the CTR mode of operation tested on an Nvidia 8800 GTX and based on both single 1 KB and four 1 KB already calculated lookup tables. The latter's equations is the same as presented before but the single 1 KB table follows the equation above:

$$e_j = k_j \oplus T_0[a_{0,j}] \oplus Rot(T_1[a_{1,j+1}] \oplus Rot(T_0[a_{2,j+2}] \oplus Rot(T_0[a_{3,j+3}])))$$

Each thread created computes its input and output address for each block and runs isolated in a single pass and produces its result. The I/O data mapping model is based on the idea of using each thread's global thread environment index as the thread's offset into the input and output buffers as follows:

```
int index = threadIdx.x + (blockIdx.x * blockDim.x);

//blockDim ⟶ number of CUDA blocks within the CUDA grid

//blockId ⟶ the current CUDA block the thread exists within

//threadId ⟶ the current thread index within the CUDA block

uint4 state = pt[index];

ct[index] = state;
```

As for the CTR mode parallel CUDA implementation, each thread works on a single AES block independently of other threads mainly for increasing occupancy. The nonce required is passed through the constant memory to the threads and the counter is calculated as the offsets above. The rekeying process is made simpler by using only one key for all data's encryption with the key schedule generated on the CPU for the reason that is serial in nature, thus a single thread should be responsible for key schedule generation. Otherwise the overhead per thread (ie. per data block) during the parallel process would be significantly high, needing each thread to generate its own schedule. The storage for the input and the output data was chosen to be the global memory between this and the texture one because it was proven to provide faster I/O process for input data along with the page lock memory usage. The lookup tables' access is one of the main bottleneck reasons so the best memory selection for their storage was examined between texture, constant and shared memory on both implementations mentioned before. The tests were conducted for two cases. One with random read memory access and one with coherent pattern. Constant memory proved to be the best solution for coherent reads while on the contrary shared memory seems to be by large margin the best choice for random access reads because it provides many ports. There were some attempts to improve the effectiveness of shared memory in order to exploit its benefits in the case of the coherent reads at first by copying a single lookup table 16 times to all the 16 shared memory banks so as to avoid memory conflicts. Although theoretically CUDA gives that opportunity for whole memory access, in practice this proves to be unfeasible. The attempts showed that the best solution was the usage of 16 X 1KB tables to save the last entry. Then, one check if the last entry is being sought and its direct value is used instead.

In the AES implementation, both types of lookup tables in combination with the GPU memories. The fastest combination was the Quad Table using shared memory which requires four tables 1 KB each to be set up within shared memory for each CUDA thread block of threads running. The implementation which is friendlier to the process above is 256 threads per thread block. The coherent shared memory read of the 1 KB table underperforms due to the additional operations which need to be done, the additional check of condition so as to source the last table entry which we mentioned above and the additional costs for each CUDA thread block memory setup process. Figure 3.4 depicts the AES CTR mode of operation using the 4 X 1 KB lookup method from which we deduce that many threads are needed to hide memory read latency. The throughput rate shown both including (15,423 Mbps) and not (6,914 Mbps) PCI-E bus transfers per kernel execution.

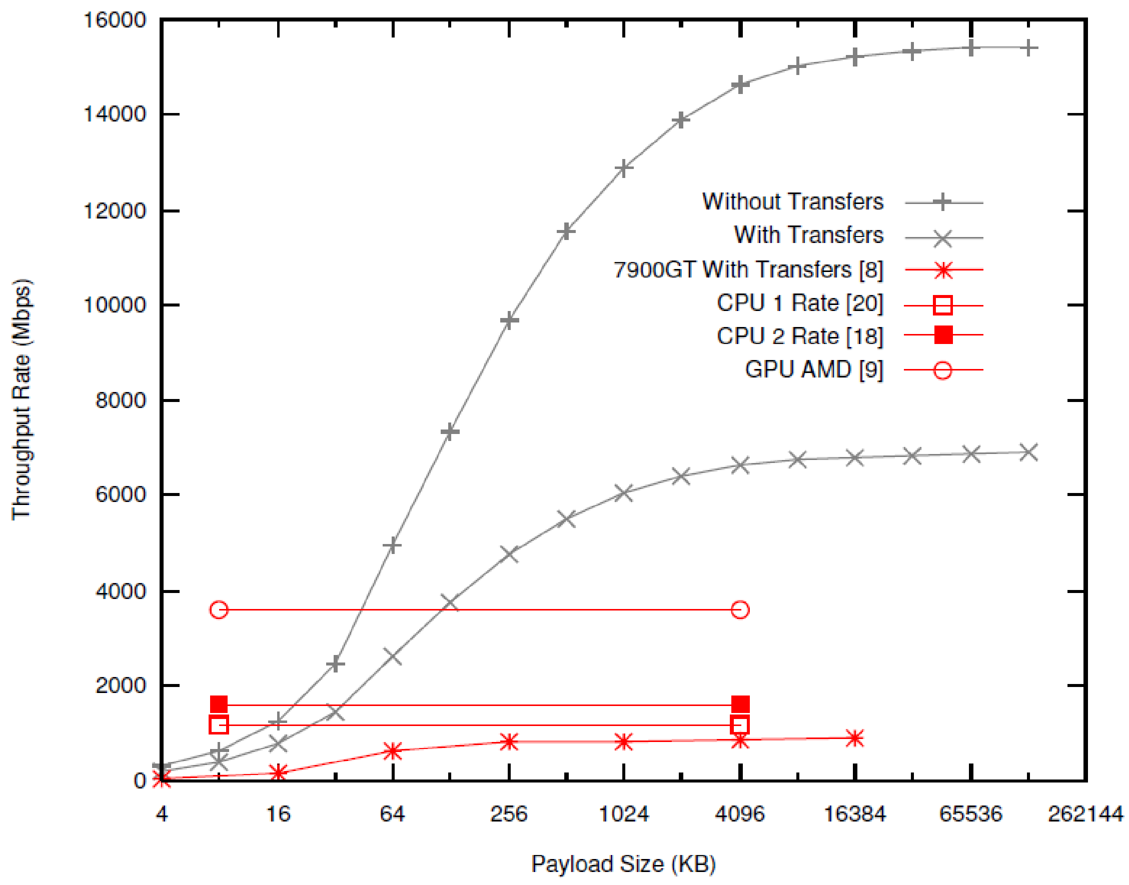


Figure 3.4: Optimized AES CTR implementation with and without data transfers[4]

We are now going to introduce the data model the researchers used in order to examine the problems that arise from mapping symmetric cryptographic services to GPU implementations alongside with its design strategy and its effect from using on the GPUs. The fundamental term used in the scheme is *payload*. Payload in this case indicates a single grouping of data which contains data for encryption along with its instructions. Client application who needs to execute a cryptographic process creates and passes the payload on to a runtime library which subsequently directs it to the appropriate implementation. One of the most crucial characteristics in which the data model was based on for its design was the ability to buffer the most possible messages that require a single stream to be processed in so that GPU could reach its full potential on performance terms. The grouping of the messages is succeeded by exposing a payload structure to the user instead of providing an API per message and the increase of the data size by making possible various cryptographic functions to be included into a single payload. The other crucial design criteria is the avoidance of pointers and the usage of offsets into the data instead. This is because the pointers pointing data would need to be transferred outside the client application so the allocated memory address space would be invalid. The scheme should also be able to allow the user to implement the underlying data and keys for reuse in a simple manner. The pseudo-code below depict the main data structures used into the scheme:

```
struct payload {
        unsigned char *data;
        unsigned char *keys;
        struct payloadDscr *dscr; };
struct payloadDscr {
        unsigned int id;
        struct keyValue *payloadMode;
        unsigned int msgcount;
        unsigned int size;
        struct msgDscr *msgs;
        struct elementDscr *keys; };
struct msgDscr {
        struct element dscr *msg;
        struct element dscr *iv;
        struct element dscr *ad;
        struct element dscr *tag;
        struct key value **msgMode; };
struct element_dscr {
        unsigned int count;
        unsigned int offset;
unsigned int size; };
```

The first struct is the main payload data structure and consists of data pointers and keys together with a payload descriptor structure used for the mapping of the messages to data and keys using an ID to identify the various payloads in an asynchronous runtime environment. The service required is defined through payload descriptor or within each message in a high level due to the variety of frameworks of different hardware devices that may use the service. The per message lower level description of the required mode is set into the "msgDscr" structure. The "element_dscr" structure contains size and count variables as the size may indicate functional differences in the used cipher. An issue that emerges when implementing a data model like the present is the I/O buffers memory allocation selection. It is imperative to use the page locked memory which requires requests to the CUDA library which then returns a pointer to the address needed and can be used in the calling process. Maximum performance requires page locked memory to be used by the input and output buffers and reuse of the same buffers as many times possible. For that reason, clients should be able to ask for input and output buffers so as to locate the allocated buffers because it is impossible for the implementation to decide which buffers should be reused independently. Thus, an encapsulating runtime, a framework like the OpenBSD Cryptographic Framework is required to map the memory allocation requests with the help of the library defining the hardware that is going to service the payload. But there are more concerns raised when it is about to bridge the this data model and particular GPU implementations. The main focus is given to a bridging layer which is designed to map the model to the specific modes of operation. In order not to have throughput issues it is important that each thread performs additional accesses, conditional branches and calculations so as to dereference the dynamic settings that have been made. These calculations can be counterbalanced with the use of the CPU as a preprocessor optimising each payload for thread parsing before being dispatched. The elements of the message descriptor require serializing on the CPU into a form which will be able to be used by each kernel thread rapidly and independently. As for identification of the instructions each thread needs, the message ID provides this information so that threads can directly seek the corresponding instructions. The key descriptor provides the access information for the key schedules and is required to create an independent schedule stream before serialization. All serialized streams mentioned are then transferred to GPU and stored into the texture memory which is the most size-flexible cache memory.

The implementation of the model contains two main processes which have to do with the message-thread mapping and the mapping between keys and messages. The former is the Logical Thread Index which is a stream produced while the messages are being serialized and makes easier locating messages using thread IDs. It is called logical due to the fact that the thread IDs assigned by the GPU and partially ascertain its physical location into the unit do not always map directly to the entries into the thread index. Physical ID plays a major role in balancing work along the GPU multiprocessors by using various assignments. This is important for serial mode of operation messages when there are not many but their size is high. The logical thread index stream contains one thread for each message of serial mode of operation and as many threads as the blocks are for parallel messages. The logical thread index contains entries which contain only IDs from the logical threads which start the messages. Figure 3.5 depicts a logical thread index and the connection with the message descriptor stream.

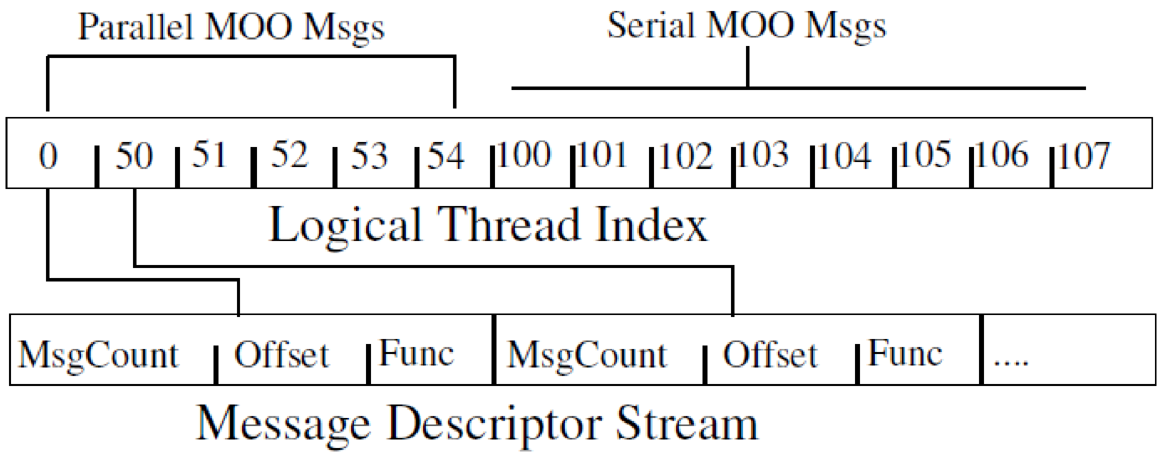


Figure 3.5: Serialised streams used by each thread for indexing[4]

Rekeying is the second crucial process of the scheme. Since key scheduler is a totally serial process, it is logical that implementing the scheduler on the CPU instead of the GPU which is suitable for parallel implementations is a better solution. When the process is completed the payload will be dispatched to the GPU. To guarantee that the keys will be reused across messages the best way possible researchers chose a hashtable cache to store the key schedules. Except for efficiency increase, this method is ideal for generating the smallest key schedule stream possible. When the application from the client side is generating the stream of the key in order to be included into the payload, it would be beneficial if the same keys had the same position into the stream. This results in fast implementation of the key schedule caching which is based on key offsets and not the key comparison. The whole mapping process between threads and messages and their data can be summarized into Figure 3.6.

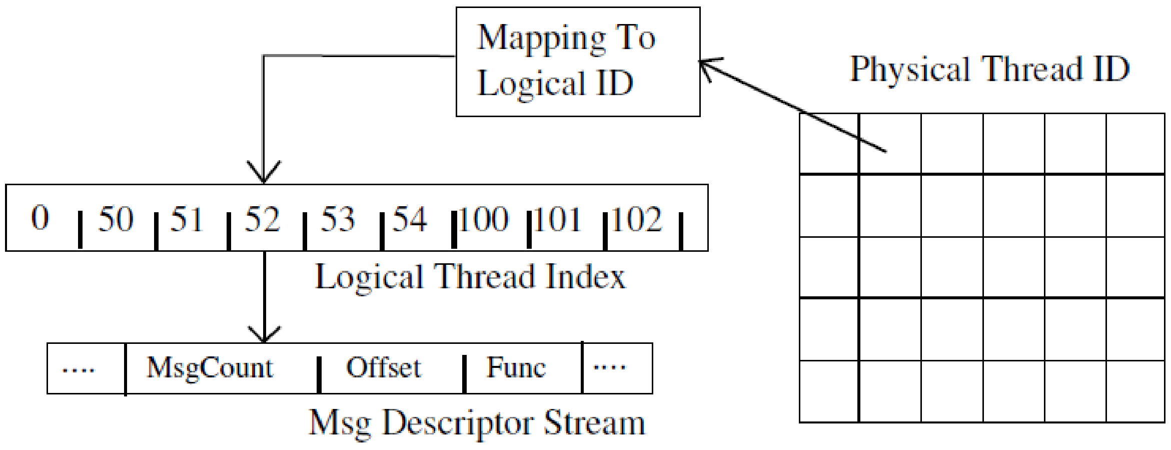


Figure 3.6: Mapping of physical threads to message IDs[4]

With regard to the throughput testings in relation to the implemented modes of operation, CBC, CFB and OFB modes for encryption were tested serially and CBC, CFB and CTR for decryption in parallel. Results in Figure 3.7 are based on CTR mode. The remarkable observation in parallel tested modes is that the performance is getting higher as the payload size increases. Note that the number of messages is in general equivalent to the number used within a single payload. This happens because resource occupancy is higher and memory latency is better hidden. On the other hand we can see that over a certain message count throughput begins to drop. It is important to mention that same buffers are reused as mentioned before so as the same key in order to imitate a single encryption session and that the experiment includes a testing comparing the 512 blocks encryption with and without rekeying. Maximum throughput observed from this in parallel implemented mode of operation was 5.810 Mbps. The increase in the overhead when using smaller messages was recorded being 16% for 16384 block sized messages reaching 45% when using messages with a length of 16 blocks. The reason for this increase is how the caching of the index stream descriptors behave on the small GPU texture memories. In the serial mode of operation message encryption the key to enhanced performance is including many messages within the payload otherwise there will be not enough threads to create a relatively high occupancy level for the GPU. The same thread to message mapping method is used and the form of the message descriptor creates a memory access pattern. Its characteristic is that threads that neighbor each other access locations of the memory which are separated by the size of the message that is processed. As shown in Figure 3.7 that pattern's impact on throughput and note that the results are based on CBC mode including an OFB CPU implementation for a more detailed comparison along with the previously mentioned parallel pattern. We observe that there is performance improvement when grouping blocks into threads which reduces the overhead for each message, this explains the large serial message count payloads having better performance than the parallel payloads. These results from the serial testings should not be compared with the AES optimized implementation as the optimized algorithm implementation is implicitly parallel.

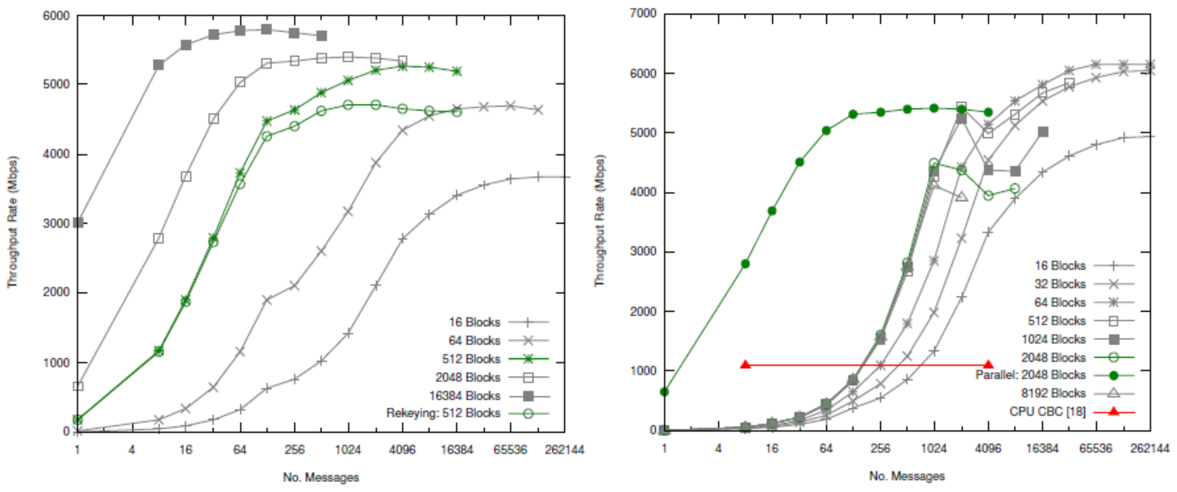


Figure 3.7: Throughput rates for parallel and serial messages respectively[4]

For larger serial messages a bottleneck is created when the number of messages increases. A possible explanation could be the access pattern. Threads that neighbor each other inside a CUDA warp use varying memory addresses for their data while message size increases. An additional separate memory test was conducted in which each thread performed some reads in sequence from the global memory starting from an offset from the thread that was previously neighbored to equal the number of the sequential reads. Figure 3.8 shows these results for varying offsets and reads in sequence in increments of blocks (16 bytes in this case). For block counts of more than 128 there is a significant drop in the performance of memory reads as the active threads increase. The reason that happens could be a combination of L2 cache bottleneck and a limitation of the number of separate DRAM available pages supported by the controllers of the DRAM resulting in concurrent reads which reduce coherency.

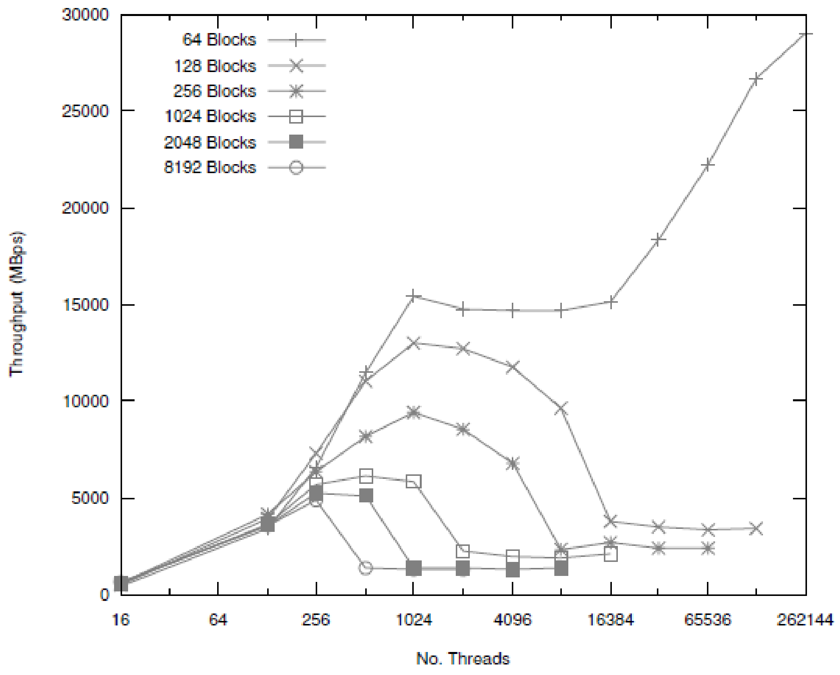


Figure 3.8: Global memory read performance with varying coherence patterns[4]

The last experiment conducted had to do with the mixing of both the modes of operation and message sizes used into one payload. This way it may be possible for the client to group all the serial messages before the payload in a simple manner. The mapping scheme used will automatically create a group of 32 messages and will distribute them evenly across the thread blocks, which will be evenly assigned to the multiprocessors available by the library of CUDA. A relevant series of tests which used to depict the effect of various mixing configurations of serial modes of operation messages across a payload was conducted for this reason. All of the payload configurations consisted of identical messages and the only change was the ordering of the messages. The differences between the scenarios depict the importance of a specific ordering of the encrypted messages when mixing messages from both types of execution into a single payload. All payloads used 960 512-block parallel messages, 992 32-block parallel ones and 1024 serial with some variations in their size. Results help us come to a specific important conclusion about what client should pay attention to when ordering; the ordering of serial mode of operation messages belonging to a payload should be their grouping into the device's SIMD width to make sure that the slots of the SIMD are occupied; the even spread of serial groups across the multiprocessors and the size-based ordering of serial messages to keep no divergent message sizes within the one SIMD grouping.

3.2 Camellia, CAST5, SEED ciphers and throughput-oriented optimization

Camellia is another symmetric key block cipher developed in Japan by Mitsubishi Electric Corporation and Nippon Telegraph and Telephone Corporation (NTT), the Japanese telecommunications company which is headquartered in Tokyo in 2001. It is suitable for both software and hardware implementations and used in low-cost smart cards and even in high-speed network systems being part of the Transport Layer Security (TLS). It has been approved by the International Organization for Standardization (ISO), the European Union's NESSIE project and the Japanese CRYPTREC project. The security level of the cipher is high enough being similar to the Advanced Encryption Standard concerning also its processing abilities.

The algorithm encrypts and decrypts with a 128 bit block size and can be used with a key size of 128, 192 or 256 bit key size. Its structure adopts the Feistel network structure and the encryption decryption process is completed in 18 or 24 rounds. The encryption process for an 128-bit key size which is the one we will examine later on is as follows: The part in which data are randomized consists of an 18-round Feistel network structure which has two function layers after the 6-th and the 12-th round and two 128-bit XOR operations before the first and the after the last round. The key scheduler generates 18 subkeys for the cipher's rounds, 4 subkeys for the XOR operations and 4 subkeys for the function layers. The decryption process as a symmetric algorithm can be done similarly to the encryption needing only the reversing of the subkeys[6].

CAST5 (also known as CAST-128) is another symmetric-key block cipher used among others as default cipher in GNU Privacy Guard (GPG) and Pretty Good Privacy (PGP) software so can be useful not only in encrypting and decrypting texts, e-mails, files, disk partitions and directories but also in signing. It was created in 1996 by Carlisle Adams and Stafford Tavares and has been approved for usage in the Communications Security Establishment by the Government of Canada. The CAST5 cipher functions with blocks sizing 64 bits and a key its size varies between 40 to 128 bits in 8-bit increment. When the key size is shorter than 80 bits the algorithm encrypts/decrypts in 12 rounds, otherwise in 16. The structure which is based on is the Feistel again and the main components of the whole mechanism are the 8 fixed S-Boxes (32-bit each) with four being used in the key scheduler and the other 4 in the encryption process.

SEED is a block cipher similar to CAST5 developed by the Korea Internet & Security Agency (KISA) and published in 1998. Each block of the encryption process has a size of 128 bits so as the key used has. S-Boxes are the key-element in this case too having 8 of 32-bit. The process is completed in 16 rounds[7].

3.2.1 Throughput oriented optimization of Camellia, CAST5 and SEED

W.-K. Lee et al.[7] conducted research on some other less known but still useful symmetric-key block ciphers based on the results from previous work on AES. These ciphers are included in OpenSSL cryptographic library and in particular are Camellia, CAST5 and SEED. Their implementation was on the highly parallelizable mode of operation CTR and was based on the algorithm from OpenSSL v1.0.1.h library version adopting a specific parallel implementation scheme which had been proposed and allowed them to encrypt many blocks even before the plaintext was available. In the experiment, each kernel of the GPU is responsible to encrypt an 16-Byte block consisting of a 64-bit counter value and a 64-bit nonce. CPU is responsible for creating the nonce using a pseudo-random number generator while the counter values with the help of thread identifier and a master counter only from the CPU. In every encryption process before the entire set of parallel threads (i.e. blocks for encryption) being launched master counter is required to be updated. The same master counter value will be assigned to each thread, which then will be added to its own thread id in order to form a unique value. The generation process of the counter value process is shown in the pseudo-code below:

```
nonce = rand()
masterCtr = 0
for i⟵0 to N do //N ⟶ total counter blocks to be encrypted
    launch NUMTHREAD of GPU threads
    //NUMTHREAD ⟶ maximum thread pools launched in every
    //iteration
    ctrBlock = (masterCtr + tid) |  | nonce
    encrypt[ctrBlock]
    return from GPU execution
    masterCtr += NUMTHREAD
end for
```

The transfer of the data and the execution of the GPU process can operate in parallel. While GPU encrypts, the blocks that have just been encrypted are copied back to the CPU and at the same time CPU will XOR plaintexts with the encrypted counter blocks. So we understand that a 3-way pipeline strategy is adopted on this implementation. This strategy is explained in detail in the scheme below:

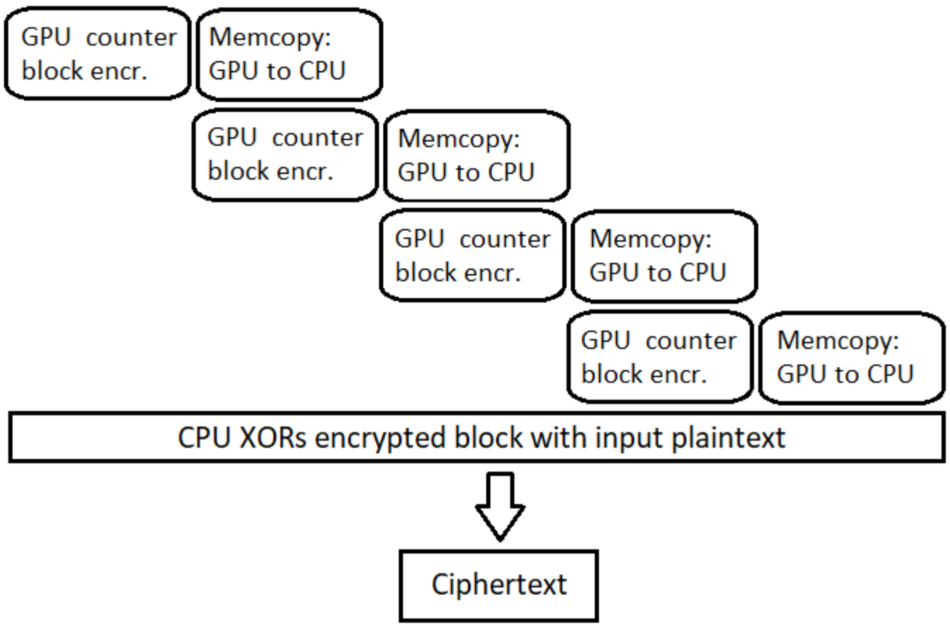


Figure 3.9: The 3-way pipeline strategy

In the memory allocation section, previous work had shown that storing keys and substitution tables in the shared memory has positive results on performance terms. This happens because the data for encryption are in most cases random which causes the random access of the substitution tables to be random too. On the other hand, storing T-boxes in shared in many cases introduces bank conflicts when we have random access but this is an issue to be examined in future work. For this reason coalesced access to global memory is in practice unfeasible if tables are stored in global or constant memory so that is why the shared memory is the best option with the storage process being started when kernel is launched so we need to load separate S-boxes and expanded keys for each thread block as shown below:

```
**if** tid < $N_t$ **then**
        table0_shared[tid] = table0_const[tid]
        table1_shared[tid] = table1_const[tid]
        table2_shared[tid] = table2_const[tid]
        table3_shared[tid] = table3_const[tid]
**end if**
```

With N_t being the number of elements in each S-box and *tid* the thread identifier which can be obtained from native instruction supported by CUDA SDK.

The coalescent access in the three algorithms can be succeeded by using a built-in vector data type which works as an array structure of array. In particular, for Camellia and SEED they used the int4 data type and for CAST5 the int2 type. This way multiple 32-bit data is accessed sort of 128-bit or 64-bit reducing multiple memory accesses into one.
The environment of the experiment is illustrated in Table 3.5. A GTX680 GPU is used with compute capability 3.0 and Kepler architecture based. Shared memory was set to 48KB and L1 cache to 16KB. High performance in encryption for block ciphers requires the GPU to be loaded in full with large amounts of data. For the CTR mode the researchers worked on it is possible to have already encrypt many counter values before the plaintext is available with this being especially useful for server environments in which the arrival time of connections cannot be predicted. Hence, their implementation was encrypting large data blocks of 256 MB based on this assumption.

Table 3.5. Specification of experiment computer

CPU	AMD 8 Core(4.4 GHz)
Memory	16 GB
OS	Windows 7
GPU Accelerator	NVIDIA GTX680
GPU Memory	4 GB
SDK	CUDA 6.0

The GPU stream mechanism which we mentioned previously was implemented for the experiment. The streams used were four. More streams did not improve the overlapping. The results for the three ciphers throughputs of the present experiment compared to other previous attempts are depicted in Table 3.6. We observe that high encryption speed can be succeeded with Camellia cipher being the fastest one with 61.1Gbps. The table includes the experiments with and without taking into consideration data transfers which as we can see cause a drop in the performance because of the transfer through PCIe.

Table 3.6. Comparison of works on throughputs achieved[7]

	Data Transfers	**GPU Used**	**Camellia**	**CAST5**	**SEED**
Nishikawa's team (2012)	W/O	Tesla C2050	50.6	N/A	N/A
Nishikawa's team (2012)	With	Tesla C2050	15.9	N/A	N/A
J. Gilger’s team (2012)	W/O	GTX295	N/A	38.6	41.5
S. Lee's team (2012)	With	GTX285	N/A	N/A	9.5
Present Work	W/O	GTX680	61.1	45.53	47.4
Present Work	With	GTX680	44.9	40.5	38.6

Due to the fact that data transfer process reduces encryption speed that much, it is imperative to only encrypt blocks in GPU and leave the XOR operation with plaintext later on in CPU. This has as a result the omission of transferring plaintext to GPU and this way data transfer bandwidth between CPU and GPU is significantly reduced. To sum up, since many block ciphers use the substitution tables T-boxes and S-boxes for permutation processes, which can use the shared memory for storage as we discussed for faster access. Furthermore, key expansion is a normally fast process, so it can be done entirely in CPU. The size of the cipher block is normally in multiples of 32-bit, so an effective option is the usage of the built-in vector data type in order to achieve coalesced memory access patterns.

3.3 An OpenSSL-based optimization attempt and benchmarking issues for few more symmetric-key block ciphers

DES was the answer of IBM to the first request ever made by the US National Bureau of Standards (NBS) in 1972 (now called National Institute of Standards and Technology-NIST) for a standardized cipher in the USA. In the 1970s the need for a commercial use of an encryption system became urgent and in 1974 a group of cryptographers from IBM proved to be the most suitable candidate. 1977 was the year when the NBS finally released all specifications of the finalized version of the cipher as the Data Encryption Standard (FIPS PUB 46) to the public. DES in the course of time developed some weaknesses due to the advance of the cryptanalysis methods so time came for an alternative solution for a cipher with federal use. Its replacement was AES in 1999 which we dealt with previously. For cases in which someone wanted DES to be implemented, a newer similar but stronger encryption scheme was proposed; the 3-DES (1995). However, we are still going to examine its performance optimization room supposing that these optimizations can be applied on 3-DES.

The DES cipher encrypts in blocks of 64 bits and uses a key of size of 56 bits. The encryption process is handled in 16 rounds (using different subkeys) of an identical operation and the structure is Feistel network in which as it happens really only encrypts and decrypts the half of the input bits in each round while the second one is copied to the next round as it is. Confusion and diffusion are the core properties of the function the cipher uses. The main stages of the algorithm are the initial (and final) permutation and the f-Function. Permutations are applied before the first and after the last round and are bitwise permutations but in fact do not make the cipher any stronger. The reason of its existence is not clear but the only logical explanation for is the arrangement of the plaintext, the ciphertext and bits in a bytewise manner in order to make the fetching of the data simpler for the 8-bit data busses which were the prevailing register size in the beginning of the 1970s. The f-Function is the most crucial element for the security of the cipher. In each round it takes the right half of the previous round’s output and the current round key as an input. The output of the function is used as an XOR-mask for encrypting the left half input bits. The function includes the use of E-box, which is used for the permutation purpose. Then, the expansion box is used to increase the diffusion behavior and finally, eight S-boxes follow which mainly provides the high security level needed for the cipher[3].

The International Data Encryption Algorithm (IDEA) is another symmetric block cipher published in 1991 by Lai, Massey, and Murphy as a modification of the Proposed Encryption Standard (PES) that was published in 1990 by Lai and Massy. PES had been designed as a replacement for the DES cipher but this did not happen eventually. However, it was included into Pretty Good Privacy (PGP). The algorithm is patented and licensed by MediaCrypt which offers a successor cipher called IDEA NXT.

IDEA encrypts in blocks of 64 bits and uses a key of 128. It consists of eight identical rounds and a final transformation round. The operations conducted are algebraic and specifically modular addition and modular multiplication[8].

Blowfish is a symmetric-key block cipher which was created in 1993 by the well-known cryptographer Bruce Schneier who wanted to provide an alternative solution to the DES and IDEA ciphers and additionally is much faster from those. It is license-free, unpatented and is available for free usage worldwide. It has proved to be resistant enough to cryptanalysis. The successor of Blowfish is Twofish which currently Schneier recommends implementing instead.

The block length is 64 bits and the key's length can be selected from 32 to 448 bits. It consists of 16 rounds and follows the feistel network. The key scheduler used is highly complex and an important characteristic of its structure is the large and key-dependent S-Boxes that it uses[9].

3.3.1 GPU-Optimized Block Ciphers in the OpenSSL Library

In this section we discuss an optimization attempt for some of the most important symmetric-key block ciphers. These are AES, DES, Blowfish, Camellia, CAST5 and IDEA which we have already briefly introduced. These ciphers will be in the form of an OpenSSL cryptographic engine. OpenSSL is a powerful, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library. The OpenSSL implementation includes tested CPU implementations of these algorithms based on which Gilger et al.[10] attempted to optimize using the GPU. It is proved that their GPU implementations can make all these block ciphers faster by a factor of ten when tested for practical application scenarios. Paolo Margara's CUDA project began to establish an Open Source portable library soa as to be used along with the OpenSSL cryptographic library which provides GPU-optimized versions of the ciphers included in the standard library. This project was the one that the algorithms' optimization attempts presented in this paper were based on. The implementation of the original project was done as a shared library) and already included the CUDA AES implementation in ECB and CBC for all key versions. In the experiment were implemented the following symmetric-key block ciphers:

- AES → Blocksize: 128 bits / Keysize: 128, 192, 256 bits
- DES → Blocksize: 64 bits / Keysize: 64 bits
- Blowfish → Blocksize: 64 bits / Keysize: 128-448 bits
- IDEA → Blocksize: 64 bits / Keysize: 128 bits
- Camellia → Blocksize: 128 bits / Keysize: 128, 192, 256 bits
- CAST5 → Blocksize: 64 bits / Keysize: 128 bits

The modes all the ciphers above were implemented with were ECB for encryption and decryption and CBC for decryption only. As we have already mentioned CBC mode cannot be implemented in parallel for encryption. These two were the only block modes available at the time in OpenSSL packages.
The implementation testing process for the researchers began with the usage of these which existed within the OpenSSL for CUDA. After ensuring their basic functionality they began the optimization attempts in order to enhance their performance. They focused on the register usage per thread, the uncoalesced memory access and divergence issues, trying a variety of optimization methods, with some of which being compiler-alternative solutions. The most crucial techniques which the researchers implemented in their experiments are the following:

- A theoretical programming flexibility was not adopted. Each GPU kernel consisted of as few parameters as possible meaning that the different modes of operations were not tested on a single GPU kernel but on one each. The same happened with the different key lengths and many other parameters too. That way a reduction in register use was succeeded and as result there was a better utilization of the SMs of the GPU.

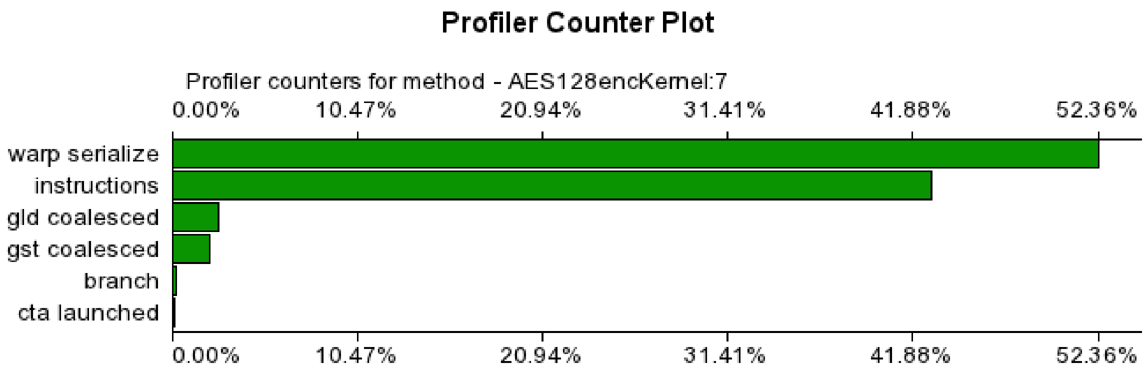
- They tried to remove all the uncoalesced memory accesses so as to avoid some memory transactions. This avoidance of all the non-aligned memory accesses for the blowfish algorithm was succeeded with the simple change in the key scheduler which is shown below:

  -
    ```
    __shared__ BF_KEY bs;          With BF_KEY structure/
    __device__ BF_KEY bsg;         misaligned mem copy

    if (TX < 18) then
    bs.P[TX] = bsg.P[TX];
    end if

    bs.S[TX] = bsg.S[TX];
    bs.S[TX+256] = bsg.S[TX+256];
    bs.S[TX+512] = bsg.S[TX+512];
    bs.S[TX+768] = bsg.S[TX+768];
    ```

  -
    ```
    __shared__ uint32_t bs[1042];      W/O BF_KEY structure/
    __device__ uint32_t bsg[1042];     aligned mem copy

    bs[TX] = bsg[TX];
    bs[TX+256] = bsg[TX+256];
    bs[TX+512] = bsg[TX+512];
    bs[TX+768] = bsg[TX+768];

    if (TX < 18) then
    bs[TX+1024] = bsg[TX+1024];
    end if
    ```

In some cases, 32-bit values not dependent on each other were retrieved from memory into a 64-bit variable and then separated manually, in such a way that the compiler would generate a single coalesced global load 64-bit instruction and not two strided global load 32-bit instructions.

- Next aim was to remove the bank conflicts which occur when using the shared memory. It was a problem that occured in the AES version in the original engine-cuda codebase. The time saved by avoiding these conflicts can be shown in Figure 3.10 cannot be ignored. The time spent due to warp serialization is clear that was a serious problem that needed to be solved.
- Register use was reduced also by a reorder of the statements. An alternative approach was to reuse variables from different stages of computation.
- Register usage can further be reduced with manual GPU loop unrolling.
- Due to the fact that many ciphers had been designed for Big-Endian CPU architectures, they thought that it was necessary to modify the key scheduler and as a result possible endian conversion was avoided.
- They used native integer functions and synchronization. They took advantage of the preprocessor by using functions which are only available on specific computing capabilities (such as 24-bit or 32-bit native multiplication).



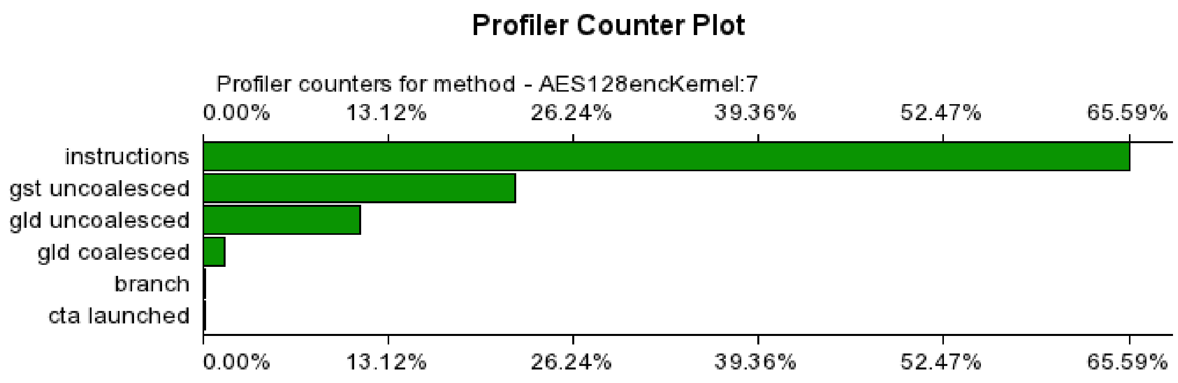


Figure 3.10: The CUDA Visual Profiler from the original engine-cuda (up) and the optimized version of AES-128 ECB (down)[10]

Table 3.7 depicts the resources used when implementing the various kernels for the block-ciphers' decryption experiments as found in nvcc while Table 3.9 the measurements from the theoretical ECB performance of the algorithms kernels, only including the time for the execution of the CUDA kernel alone. Register use is in relation with each thread and there was a count on the shared memory per block and for the whole kernel constant memory. The modifications made for optimization had to do with the memory transfer functions, graphics card context (the way it was set up) and the memory allocation and they did not perform any stage of the key-scheduler on the GPU which was performed by the CPU instead in order to ensure that the implementation was correctly done by comparing its output with the CPU from the stock OpenSSL. The benchmark system has the specifications that are shown in the table below:

Table 3.7 Specification of experiment computer

CPU	Core i7 960 (3.20GHz)
Memory	12 GB DDR3 RDIMM
OS	CentOS 5.3 (kernel ver2.6.18)

HDD	Intel X-25 M II SSD (160GB)
Compiler	gcc ver 4.4.6
GPU Accelerator	NVIDIA Geforce GTX 295 (CC 1.3)
Kernel	Linux 3.0.0-17-generic x86_64
CUDA	CUDA toolkit 4.1
Driver	NVIDIA UNIX x86_64 285.05.33
CUDA Compiler	GCC 4.4.6

The GeForce GTX 295 this experiment's system consists of two independent GPUs on two different PCBs (printed circuit boards) with each of them being considered as an independent GPU to the system. The researchers used one of these GPUs for their work. CPU reference speeds were acquired using OpenSSL v1.0.1 and not the four but the one CPU core of the system. Theoretical kernel execution speed does not take into account the transfer time between CPU and GPU, while the measurement of the practical encryption speed includes these transfers.

Table 3.8. Memory consumption for each cipher decryption process[10]

Cipher	**Mode**	**Registers**	**Shared Memory**	**Constant Memory**
AES-128, AES-192, AES-256	ECB	13/14	4376 bytes	264 bytes
AES-128, AES-192, AES-256	CBC	15	4384 bytes	264 bytes
Blowfish	ECB	10	4176 bytes	8 bytes
Blowfish	CBC	12	4184 bytes	8 bytes

DES	ECB	9	2056 bytes	136 bytes
DES	CBC	10	2064 bytes	136 bytes
CAST5	ECB	10	4104 bytes	144 bytes
CAST5	CBC	12	4112 bytes	144 bytes
Camellia	ECB	14	4104 bytes	296 bytes
Camellia	CBC	14	4112 bytes	296 bytes
IDEA	ECB	10	224 bytes	216 bytes
IDEA	CBC	12	232 bytes	224 bytes

Table 3.9. Kernel performance for ECB encryption process[10]

Cipher	**RND_B Kernel ms**	**RND_B MB/sec**	**Z_B Kernel ms**	**Z_B MB/sec**	**Δ**
AES-128	2.16	29613	1.36	47021	1.59
AES-192	2.57	24922	1.58	40461	1.62
AES-256	2.97	21572	1.87	34234	1.59
Blowfish	2.12	30249	1.49	43077	1.42
DES	4.14	15460	2.75	23279	1.51
CAST5	2.19	29203	1.54	41519	1.42
Camellia	2.43	26302	1.66	38647	1.47
IDEA	1.75	36512	1.71	37388	01.02

In the last table the theoretical performance can be derived from the measured time. The importance of the payload data is highlighted by measuring the performance with pseudo-random data (using /dev/urandom) and zero-bytes and the usage of Δ value to indicate the speed ratio of zero over random bytes. Random bytes are referred to as RND_B in the table above and zero-bytes as Z_B. The payload data is used to examine the memory access for specific operations (e.g. in a lookup table). When the payload contains only uniform bytes, the same memory area will be accessed for each one byte of payload data, resulting in great performance for constant memory (after being cached after the first access) but when the data exhibits a certain amount, new memory areas will be asked for in each request so then shared memory is clearly the best solution.

3.3.2 OpenSSL benchmarks and how to reproduce them

In order to measure the gain from the GPU implemented block ciphers, the researchers used the OpenSSL speed command, which is used as a scheduler for runs on increasingly large blocks consisting of zero-byte data and this is how the achieved throughput is measured. The results derived from the mean value of five consecutive runs and in order to have an accurate comparison to the CPU only implementation the measured time includes the transfer of the data from CPU to the GPU, the GPU kernel execution and the transfer back to host. In Figure 3.11 the results of GPU and CPU benchmarks for ECB of each cipher are depicted. As the launch of a GPU kernel causes a significant latency along with the memory transfers the GPU outperforms the CPU implementation only when the payload data is large enough and in the specific experiment larger than 16KB.

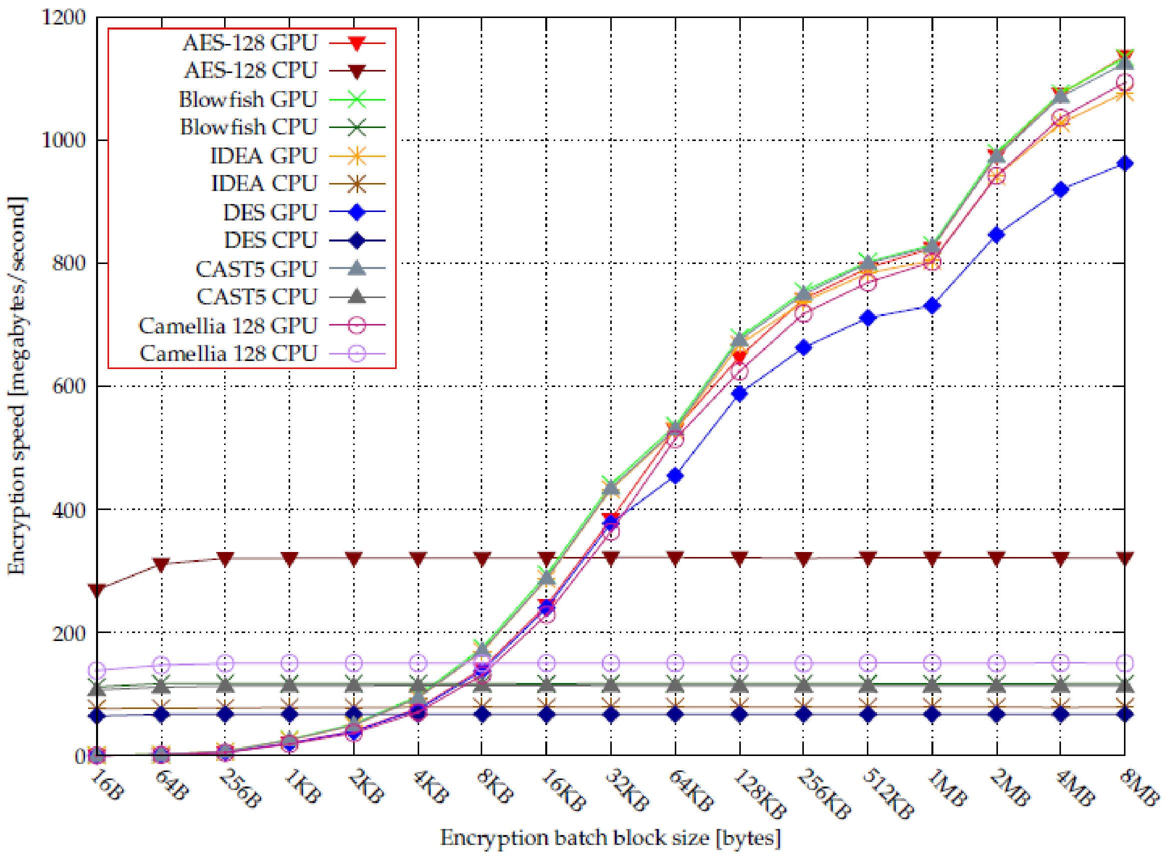


Figure 3.11: ECB encryption ciphers comparison with CUDA on one PCB of a GeForce GTX 295 (OpenSSL speed)[10]

We are now going to provide a number of benchmarking details which are required for the production of reproducible comparable benchmark results and we present an overview on the details as guidelines for future implementation efforts which can be valuable not only for the field of cryptography.

- **Framework and kernel**. For different frameworks, the host operations and memory transfer time make the actual kernel execution time seem insignificant.
- **Structure of payload**. The benchmark of the speed of the encryption process is affiliated with the kind of data being processed. Zero-byte files may provide ease for generation and reproducibility but they can provide an apparently good result.

Block ciphers should be developed not only using zero bytes but also random data, and all publications should clearly show the source of the payload data. If anything the storing of read-only lookup tables should be decided after taking into account the process behaviour with random data.

- **Correctness issues.** Since GPU programmes are difficult to debug, when conducting an experiment the cipher's code should be totally correct. As a result a combination of key sizes, different keys payload length and structure should be tested multiple times. A typical reason for an execution failure is the use of a payload with size not multiple of the thread block configuration.
- **Scheduling priorities**. There is an option in CUDA that gives the choice of spinning for waiting on the return of the kernel, which for many repeated small calls can cause a significant difference. Verification of the method to poll for the GPU kernel is very important.
- **Side effects elimination**. When benchmarking, GPUs need to be switched to compute a specific mode and if any X server runs at the same time it should be stopped. Benchmarking cryptography means that large blocks of pinned memory need to be allocated on the host side, so the system's RAM remaining needs to be enough. The CPU should only be involved in the execution of the host-thread. It is imperative to use the GPU driver which is supplied by the vendor in cooperation with the modern GPU's framework. Other drivers will possibly work properly too but will probably perform much worse when used with CUDA.
- **Usage of reference implementations**. A publicly accessible and already established CPU implementation of a block cipher should be used when comparing to a GPU implementation. Some specifications for the CPU we refer to need to be clarified too. For example, the number of cores used, whether they are used native instructions customized to suit the CPU platform or special instructions like AES-NI.

In contrast to programs running on CPUs, in GPU programming the application area before the actual implementation area in which the application runs needs to be carefully considered beforehand. There are some possible scenarios in which possibly GPU assisted symmetric-key cryptography could be used in the future and point out the unique characteristics of each. First of all, it can be used in key breaking methods like dictionary attacks or brute-force ones. This process requires the use of independent threads which work with a distinct key each and can signal success of the attack using the global memory of the GPU. Block ciphers that use a large key-dependent S-Box e.g. the Blowfish cipher will not have the same enhanced performance as other ciphers will do, since the large schedule elements should be stored in the limited fast memory space for the breaker to work efficiently. Another usage can be the SSL acceleration. Many websites use the slow and partial HTTP protocol so GPU cryptography could cheaply accelerate the SSL operations. Some options to use specific settings GPUs are the following:

- **Built in**. The server software itself is linked against CUDA or other interface and directly calls device code to execute the cryptographic operations.
- **Library.** A more functional solution is a widely used cryptographic library (like OpenSSL or libgcrypt) which is the initial target for GPU acceleration because all the software using it would benefit instantly.
- **Standalone**. A standalone mechanism linked against CUDA which can interact with the user and the back-end software. This approach is extremely flexible because this way any changes are not required to be done to the server software and can be scaled independently of it.

Disk encryption is another critical sector in which GPU cryptography can be used. Disk encryption acceleration can only be succeeded if the write speed of the disk surpasses the cryptographic performance of the CPU which is in general not true for modern consumer systems. Including AES circuitry in CPUs and often directly inside the hard drive shows that there are plenty of options for securely storing data on devices like these. Last but not least, operating systems such as the kernel of the Linux OS can exploit the GPU accelerated cryptography. Kernel cryptography services are crucial for some software and dm-crypt for disk encryption is an example. These days the usage of GPU from kernel space requires a search in the CUDA libraries in user space and then a return to the kernel. These processes require two costly context switches. So we understand that is an area which needs further investigation for performance enhancement.

CHAPTER 4

PUBLIC-KEY CIPHERS-RSA OPTIMIZATION

4.1 The RSA cryptosystem

In 1977, Ronald Rivest, Adi Shamir and Leonard Adleman proposed a scheme which currently is the most widely used asymmetric cryptographic scheme. This was the RSA cipher (Rivest–Shamir–Adleman). The algorithm was patented only in the USA until 2000. RSA has many applications but the most common ones are the encryption of pieces of data (usually in key transport) and digital signatures/digital certificates. Despite its many advantages, it is not supposed to replace symmetric ciphers since in many cases is slower than some of them like AES and we must not forget that performance is also very important when we need to decide which cipher to implement. This happens because of the many computations that take place in the cipher and we will briefly present shortly. So the main purpose of this encryption scheme is to work alongside a symmetric key cipher and securely exchange a key with it (the key transport we mentioned before). The whole algorithm's mechanism is based on the one-way function of the integer factorization problem which indicates that it may be very easy to compute the multiple of two large primes but the factoring of the resulting product is very hard to find. Euler has played a major role in the design of the algorithm since his theorem and his phi function are fundamental for its function. RSA encryption and decryption process is done in the integer ring Z_n (bit string representing the plaintext is considered to be an element in $Z_n = \{0, 1,..., n - 1\}$) and some modular computations are very important for the cipher. Since the process is done in this integer ring, the binary value of the plaintext and the ciphertext must be less than n. The encryption and decryption process are shown below[3]:

Encryption: Given the public key $(n, e) = k_{pub}$ and the plaintext x, the encryption function is the following:

$$y = e_{k_{pub}}(x) \equiv x^{e} \bmod n$$

where $x, y \in Z_n$.

Decryption: Given the private key $d = k_{pr}$ and the ciphertext y, the decryption function is the following:

$$x = d_{k_{pr}}(y) \equiv y^{d} \bmod n$$

where $x, y \in Z_n$.

We should also mention that the key generation is a very complex process that mainly uses the greatest common divisor and computes the private key with a modular computation. This process simplified is shown below:

<u>Output:</u> public key: $k_{pub} = (n, e)$ and private key: $k_{pr} = (d)$

1. Selection of two secret large primes p and q (approximately same size but not too close).
2. Computation of $n = p * q$.
3. Computation of $\Phi(n) = (p - 1) * (q - 1)$.
4. Selection of the public exponent $e \in \{1, 2,..., \Phi(n) - 1\}$ such that

   $gcd(e, \Phi(n)) = 1$.
5. Computation of private key d such that $d * e \equiv 1 \bmod \Phi(n)$

4.1.1 The expensive modular operations of RSA

As we mentioned in the presentation of the algorithm above, RSA uses two different keys for encryption and decryption. The public key which both parties who take part in the process know is used for encryption, while for decoding is used the private key. The most important operations are the modular ones and this is the reason why the whole process is computationally expensive. In general, if we want to implement an efficient RSA algorithm, the key size chosen should be small but lamentably, in this case many security issues arise. This process is called modular exponential and is defined by the equations below:

$$(u + v) \bmod m = ((u \bmod m) + (v \bmod m)) \bmod m$$

$$(u - v) \bmod m = ((u \bmod m) - (v \bmod m)) \bmod m$$

$$(u * v) \bmod m = ((u \bmod m) * (v \bmod m)) \bmod m$$

There are some different ways that these operations can be operated then they need to be done repeatedly. The simplest but also naive way to complete them is by performing *e-1* modular multiplications. As we understand, this method is definitely not efficient because for large plaintext encryption a large amount of modular multiplications are required. A good example just to understand how expensive this method is, is the case in which we have g=4, e=10 and m=497. In order to reach to the desirable result (c=403) we need to apply the following operations:

1. $e = 1, c = 4 \bmod 497 = 4$
2. $e = 2, c = 4 * 4 \bmod 497 = 16 \bmod 497 = 16$
3. $e = 3, c = 16 * 4 \bmod 497 = 64 \bmod 497 = 64$

.

.

.

10. $e = 10, c = 225 * 4 \, mod \, 497 = 900 \, mod \, 497 = 403$

A much more effective way to complete the modular exponentiation than the one above can be used if e is even. In this case the modular exponentiation can be calculated as $g^{e} \, mod \, m = (g^{e/2} * g^{e/2}) \, mod \, m$ and that way reduces the number of multiplications. There are two forms of the algorithm with which it can be implemented, the right-to-left binary modular exponential and the from left to right binary modular exponential. Another multiplication algorithm which is similar to the left-to-right binary modular exponentiation is the left-to-right k-ary exponentiation. The difference here is that in each iteration more than one bit of the exponent is processed and works even more efficiently when computations that have been concluded are performed in advance and are used again and again. The last efficient method for performing these operations is the sliding window exponential. The interesting thing behind this method is the less pre-computations needed as compared to the algorithm we just referred to and and as a result the desirable reduction in the average multiplications number needed for computation[11].

4.1.2 Parallelizing RSA modulo function methods

In this section we present the structure of the parallelized scheme created by Mahajan and Singh[11]. In general, the host and device code basically follow the following steps:

1. CPU accepts the values needed for the message and all the key parameters.
2. CPU allocates memory on the CUDA supported available GPU and copies the values on the device memory.
3. CPU launches the CUDA kernel on the device.
4. GPU encrypts each character of the message using the RSA algorithm using as many threads as the message’s length.
5. Control is transferred back to the CPU.
6. The results from the GPU are copied and displayed from the GPU.

Very large power of numbers are not supported by built in data types hence a special technique was adopted to achieve the modulo calculation. The adopted principle is the following:

$$C = M^{e} \, mod \, n \equiv (M^{e-x} \, mod \, n \; * \; M^{x} \, mod \, n) \, mod \, n$$

So by iterating over a suitable value of x the desired result is derived.

CUDA Kernel pseudo-codes:

```
Input: __global__ void rsa(int *num, int *key, int *n, unsigned int *results)
//executed on the device and called from the host
Output: $num^{key} mod n$

Declare int vars: temp, total_threads
i ← threadIdx.x + blockDim.x * blockIdx.x
if ($i < total\_threads$) then
    temp ← mod(num[i], *key, *n)
    atomicExch(result[i],temp)
end if
```

Pointers for variables used because rsa() is executed on the device, so variables must point to device memory. So memory space on the GPU needs to be allocated.

```
Input: __device__ long long int mod(int g, int e, int n)
//executed on the device and called from the host
Output: $g^{e} mod n$
```

```
Declare vars: a, ret, size
unsigned int a ← (g%n)*(g%n)
unsigned long long int ret ← 1
float size ← (float)e/2
if ($e == 0$) then
    return (g%n)
end if
else
    while (true)
        if ($size > 0.5$) then
            ret ← (ret*a)%n
            size ← size-1.0
        end if
        else if ($size == 0.5$) then
            ret ← (ret*(g%n))%n
            break
        end if
        else
            break
        end if
end if
return ret
```

The integration of CPU and the GPU (host code) basically follows the steps below:

- Checking for the available GPU cards

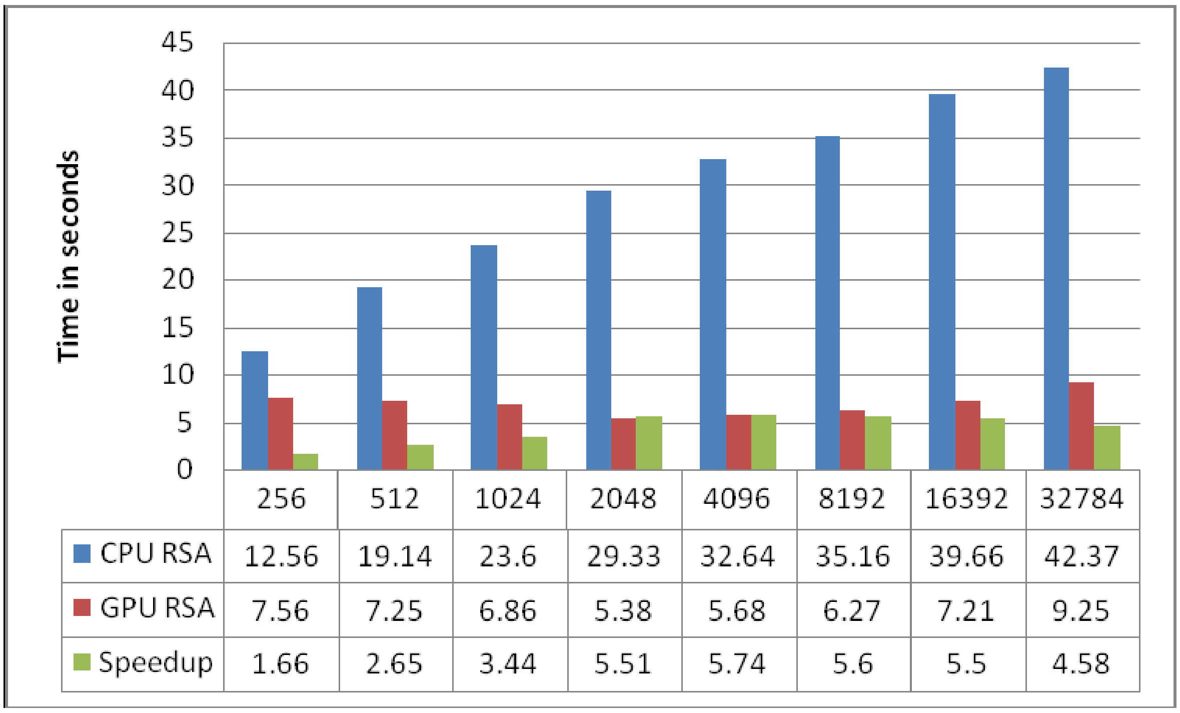
- Creating copies of variables for device and host
- Using cudaMalloc() in order to allocate memory space for device variables
- Setup of the input values
- Copying input values to the device using cudaMemcpy function.
- Launch of the kernel on GPU with the following parameters: rsa<<<num_blocks, num_threads>>> (dev_num, dev_key, dev_den, dev_res)
- Copying result back to CPU with the cudaMemcpy function

The experiment consists of three parts with the first one the traditional algorithm is tested on CPU for small prime numbers, the second runs the parallelized CUDA RSA algorithm on GPU for small prime numbers again and then is compared with the first one. The last test is on GPU too but in this case for large prime numbers which are not supported by the built-in data types of the CPU RSA. As regards the test environment, the algorithms were tested for message values between 0 and 800 supporting the complete ASCII table this way and for 8-bit key values (the calculation of ciphertext was implemented in parallel on an array of integers). Table 4.1 provides the specification of the computer used for the testings:

Table 4.1. Specification of experiment computer

CPU	Intel(R) Core(TM) i3-2370M(2.4GHZ)
Memory	4 GB DDR2(2GHZ)
OS	Windows 7
GPU Accelerator	Nvidia GeForce GT 630M
GPU Memory	512MB
CUDA	CUDA v5.5

Figure 4.1 shows the first comparison we mentioned before. It is between the CPU and GPU implementations for small prime numbers and along with them the actual speedup is depicted.



	256	512	1024	2048	4096	8192	16392	32784
🟦 CPU RSA	12.56	19.14	23.6	29.33	32.64	35.16	39.66	42.37
🟥 GPU RSA	7.56	7.25	6.86	5.38	5.68	6.27	7.21	9.25
🟩 Speedup	1.66	2.65	3.44	5.51	5.74	5.6	5.5	4.58

Figure 4.1: The effect of data input on CPU and GPU for RSA[11]

The next measurement had to do with the GPU implementation for large numbers of n. We can see that the relation between the execution time and the amount of the input data is linear for a certain amount of input. The execution time varies when using different numbers of thread blocks and threads for data input but the clear observation is that with the increase of the data size, the execution time will be much shorter according to the number of threads used. In particular, the data size increase from 1024 to 8192 bytes along with the drop in execution time proves that the more the data size is, the more the algorithm is parallelized, and the time spent is less. All these details are shown in Table 4.2.

Table 4.2. GPU implemented RSA for large prime numbers and large value of n (n = 1005 * 509)[11]

Data Size (bytes)	Number of thread blocks	Threads/thread block	Execution time
256	8	32	6.08
512	16	32	6.52
1024	32	32	6.69
2048	64	32	5.53
4096	128	32	6.58
8192	256	32	6.66
16392	512	32	7.81
32784	1024	32	8.76

The great enhancement in performance caused by GPU acceleration is demonstrated in Figure 4.2 in which the comparison between CPU implementation for small value of n (137*131) and GPU implementation for large prime numbers and value of n(1009*509) is made.

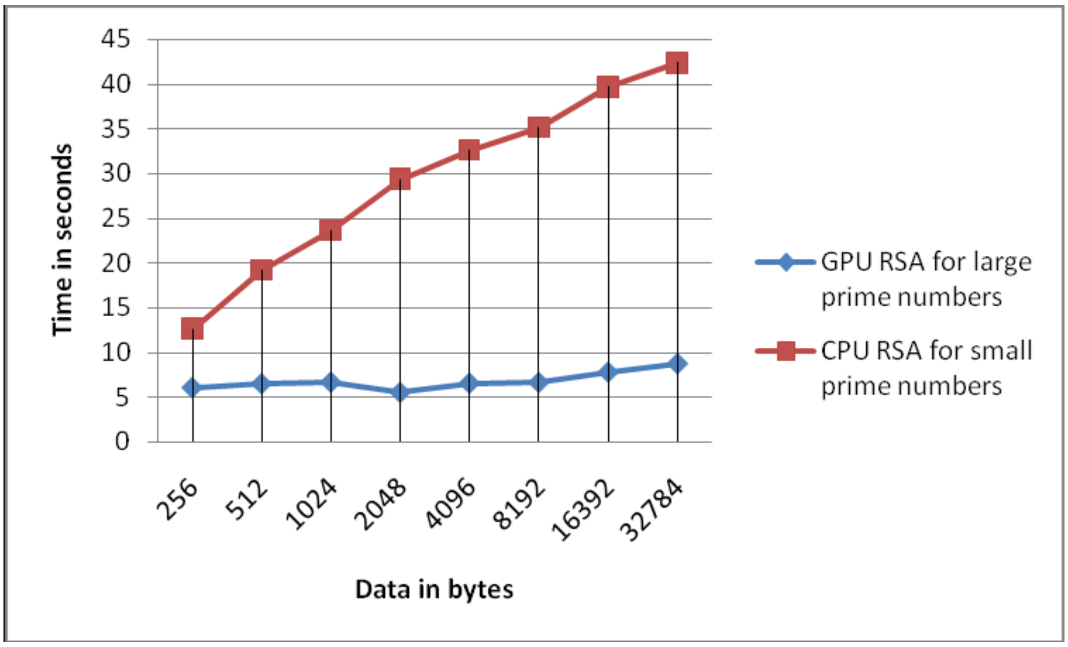


Figure 4.2: Comparison of CPU RSA for small prime numbers with GPU RSA for large prime numbers[11]

Now we are going to present a hybrid scheme for RSA proposed by Fadhil and Younis[12]for multicore CPUs and GPUs and for variable key size. In order for the performance enhancement to be highlighted, three different implementations of the algorithm were conducted and then compared to the existing Crypto++ library' sequential implementation. The basis for this optimization attempt is the Montgomery algorithm which despite being an infrequent algorithm in public-key cryptography, provides interesting efficiency for modular multiplications and exponentiation operations when the modulus is large (1024 bits at least). It was designed by Peter Montgomery in 1985. The algorithm has two parts, the multiplication and the reduction. The multiplication process is a technique for computing a*b mod n for positive whole numbers a, b, and n. It reduces the execution time needed in cases in which there are large numbers of multiplications that need to be conducted using the same modulus n, and with a small number of multipliers. The key for this method's success is that it needs a number of multiplications much less than n by successively squaring and multiplying according to the arrangement of the bits in the binary expression for n. Something that is crucial for the montgomery multiplication implementation method is that if in the operation $a^b \, mod \, n$, *a* and *b* are less than modulus *n* we need to declare an extra integer *r* such that $r > n$ and $gcd(r, n) = 1$changing, in essence, the reduction modulo from *n* to *r*. As a consequence, *r* is just a masking operation. If *r* is a reduction modulo power of 2 numbers, *n* should be not an even but an odd number, in order to satisfy the greatest common divisor requirement. The computation of *MonMul(a,b)* is explained in the pseudo-code below:

<u>Montgomery multiplication:</u> ($MonMul\,(a', \, b') \; = \; a' \, . \, b' \, . \, r^{-1} \, (mod \, n)$)

Input: An odd modulus n and a radix

Output: a' .b' .r-1 (mod n)

$r = 2^{\lceil log_2 n \rceil}$, such that $gcd(n, r = 1)$, a supplementary value $n' = - n^{-1} \, mod \, r$, 2 n-residue integers a' and b'.

```
MonMul (a', b')
$t := a' . b'$
$u := (t + [t. n' \, mod \, r]. n) / r$
**if** ($u \geq n$) **then**
        return (u-n)
**else**
        return u
**end if**
```

a and b are numbers that represent the n-residues, which are $a' = a . r \, mod \, n, \; b' = b . r \, mod \, n.$

Both integers r^{-1} and n' are calculated, by using the Extended Euclidean algorithm, such that: $rr^{-1} - nn' = 1$.

The result of the multiplication in the n-residue is:$u' = a' . b'. r^{-1} \, mod \, n$.

The final stage in the procedure is the conversion that needs to be done in order to transform the result back to the normal residue representation:
$u = MonMul \, (u', 1)$

The modular exponentiation $x = a^{e} \, mod \, N$that uses the Montgomery multiplications in order to be computed and the algorithm is shown below:

<u>Montgomery Reduction Algorithm:</u>

```
Input: $a, e$ and $n$
Output: $a' . b' . r^{-1} \ (mod\ n)$

MonExp(a, e, n)
$a' = a . r\ mod\ n$
$x' = 1 . r\ mod\ n$

for ($i = n - 1$ to $0$)
          x' = MonMul(x', x')
          if ($e_i == 1$) then
                    x' = MonMul(x', a')
          end if
end for

x = MonMul(x', 1)
return x
```

The whole RSA implementation was done for four different forms in order to provide a better comparison among them. The first was the Crypto++ library algorithm for CPUs (with a standard 1024-bit key size), the second a sequential one using the Montgomery method we presented above for CPU again, the third a parallelized one for multithreaded CPUs and the last one a parallelized one for GPUs (the last three implementations work for variable key size).

The decision for the type of the variables that is needed for the whole encryption process is a critical issue when the algorithm's programme was written. Here we should mention that the different implementations proposed are implemented using **C#** programming language and **GPU.net framework**. The representation of large numbers as 1024 bits or higher for the key generation algorithms parameters required the usage of the Biginteger Class. The main bottleneck of the encryption process is caused by the large size of data so the ideal scenario in the parallel RSA model implementation would be no dependencies between the data to exist. To succeed this, data had to be divided into smaller pieces with each piece being calculated by a thread and specifically independently conducting a modular exponentiation. We only present here the algorithm for the parallel GPU implementation as this is the point of interest of this diploma thesis:

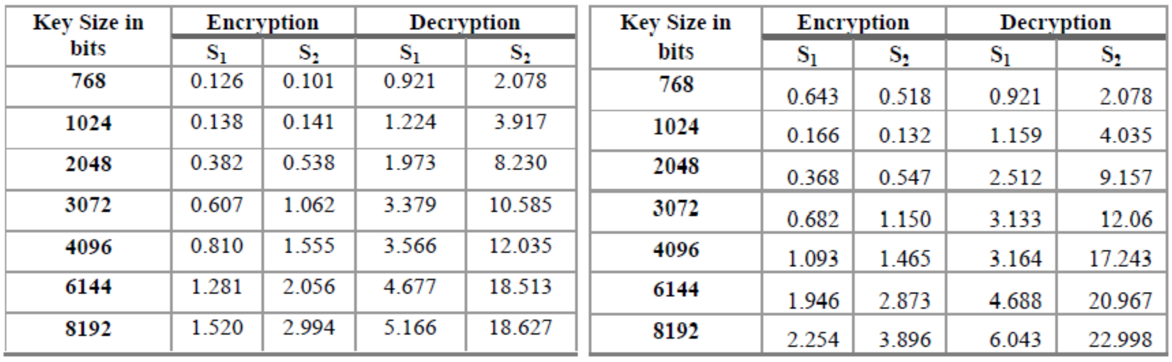
Parallel RSA pseudo-code for GPU:

```
Public key {e,n}                         // Keys Generation
        public struct RSA_Public_Key
Private key {d,p,q}
        public struct RSA_Secret_Key

Input: Text for encryption

Launcher.SetGridSize (512);   // Set kernel launch parameters
Launcher.SetBlockSize (128);

Reduce_GPU (A, n, m, mPrime);       // call of the kernel method

Int ThreadId = BlockDimension.X * BlockIndex.X +
ThreadIndex.X;
Int TotalThreads = BlockDimension.X * GridDimension.X;
```

The evaluation of the performance enhancement was conducted by using the metrics of the speed up factor. This factor for parallel computation working on *p* processors is the ratio $S_p = \frac{T_s}{T_p}$, where T_s depicts the execution time needed while performing computation on a single processor and T_p the execution time taken to perform the exact same computation using *p* processors. In order to ensure that the speed up factor is fair for the parallel version, researchers considered the sequential time of their sequential version. Latency and throughput are two factors that had to be considered in the evaluation process. Taking all the aspects above into account, researchers conducted the experiment using three test groups:

- **Group 1**. In this group, the message size is fixed to 760 bits. It was encrypted and decrypted with a varied key size from 768 to 8192 bits.
- **Group 2**. In this group, the input messages varied in size that is convenient with the size of the key used in the encryption process (one byte less than modulus size).
- **Group 3**. In this group, the thread block size was selected to vary so as to be multiple of message size in steps of 50 to 600. This is the test which is crucial in determining the speed up gain as far as the throughput is concerned.

The specification of the computer used is an Intel Core I7-2670QM(2.20GHz) CPU, a RAM of 12GB memory space, a NVIDIA GeForce GT630M GPU consisting of 96 cores and an operating system of Windows 7 Home Premium. In the following figure of tables (Figure 4.3), the first table derives from the implementation of test group 1 and the second from the test group 2. They depict the speed up factor and specifically S_1 is the speed up factor for the multi-threaded CPU implementation and S_2 the factor for the GPU implementation. The first table has to do with the measurements done for The execution time in milliseconds for encryption and decryption of 760 bits message length with different key size while the second table with the latency for encryption/decryption of varied message size and variant key size. The measurements showed that Crypto++ and the sequential implementation do not have a significant difference in performance. As regards the execution time, it seems that the GPU implementation begins to be faster than the other two when the size of the key is 3072 bits or higher. It was also observed that the time needed to decrypt a message was more than the encryption time needed. The reason for this, is that the public exponent *e* used is smaller than the private exponent.In general, in the execution time the GPU exceeds the other two implementations for CPU for all the key sizes; A conclusion we can also reach to by studying the results is that the GPU is more powerful when undertakes heavy computations.



Key Size in bits	Encryption		Decryption	
	S_1	S_2	S_1	S_2
768	0.126	0.101	0.921	2.078
1024	0.138	0.141	1.224	3.917
2048	0.382	0.538	1.973	8.230
3072	0.607	1.062	3.379	10.585
4096	0.810	1.555	3.566	12.035
6144	1.281	2.056	4.677	18.513
8192	1.520	2.994	5.166	18.627

Key Size in bits	Encryption		Decryption	
	S_1	S_2	S_1	S_2
768	0.643	0.518	0.921	2.078
1024	0.166	0.132	1.159	4.035
2048	0.368	0.547	2.512	9.157
3072	0.682	1.150	3.133	12.06
4096	1.093	1.465	3.164	17.243
6144	1.946	2.873	4.688	20.967
8192	2.254	3.896	6.043	22.998

Figure 4.3: Speed up factor for test groups 1 and 2[12]

Studying the results from test group 3, turns out that the encryption for one message using a 1024-bit key takes 0.75 ms for the sequential execution and we would expect that for the same type of execution for 600 messages would be needed 0.75*600=450 ms but “surprisingly” from the tests of group 3 it was proved to need 1262.272 ms. The obvious reason for this unexpected latency is more time taken because of the looping overhead. As regards the multi-thread CPU implementation it can be derived from the results that for 600 messages a 540 ms execution time would be expected but what happens is an enhancement in performance with 439.475 ms taken to encrypt the messages because of the free resources available for the computing process that are occupied by all the available threads. GPU implementation results give a similar observation. The encryption of 600 messages is 1.572 times faster. The ability of the GPU to complete heavy computations with large data is depicted also by the decryption process in which the GPU implementation is almost 14 times faster and the multithreaded CPU model 2.17 times faster. The last and very important measurement taken had to do with the throughput gained from GPU implementation in comparison to the sequential and multithread ones. The gain from 1024-bit key size was much bigger than from the 2048 bits. For the 1024 bits the gain was approximately 1800 msg/sec while for 2048-bit key approximately 250msg/sec and the reason for that is the overlap of multi-thread operation whenever the free resources are available for use.

4.1.3 An alternative Montgomery approach and programming techniques

A different approach to the existing Montgomery multiplication method for RSA performance enhancement was presented by Neves and Araujo[13]. In fact, these methods are not new to the researchers of this field but they had been neglected in the literature until that time. Except for these Montgomery alterations, the researchers applied and presented some programming techniques that generally enhance the parallel performance of the programmes. These mainly were loop unrolling and PTX assembly. They attempted to exploit the best way possible the provided tools and the hardware specifications and select algorithms better suited to the architecture. They made their experiments on the 1024-bit key sized RSA as this is the most common key implementation and it allowed them to make easier comparisons and draw more exact conclusions concerning the existing GPU and CPU implementations.

The available tools and hardware were exploited by performing an extended manual unrolling, using the GPU's PTX assembly tool directly and they managed to maximize the register use to the detriment of less threads per thread block. The algorithms selected that better suited the architecture were three generally untried Montgomery reduction approaches from which the last two proved to be very effective. Coarsely Integrated Operand Scanning (CIOS), Finely Integrated Operand Scanning (FIOS) and the Finely Integrated Product Scanning (FIPS). FIOS and FIPS were more suitable for their purpose compared to CIOS because both are methods which precisely integrate both reduction and multiplication in the same loop. A technique in which all of these algorithms were based on was the Chinese Remainder Theorem (CRT), a theorem that Quisquarter first implemented in security schemes and according to which we can replace one large 1024-bit (RSA-1024) exponentiation by two smaller 512-bit independent ones. On mathematical terms, the CRT says that[14] assuming we have two coprime numbers *p,q*, the system of two equations $x = a \, mod \, p$and $x = b \, mod \, q$, has always a unique solution for $x \, mod \, pq$. As a result, the reverse direction we are interested in implies that given a number $x \in \boldsymbol{Z}_{pq}$we can reduce $x \, mod \, p$ and $x \, mod \, q$ in order to obtain two equations of the system's form above.

An important decision that needs to be made when multi-precision arithmetic is about to be implemented on a specific architecture, has to do with the representation of the numbers because not all representations can exploit the specific features of each GPU architecture. In the present experiment researchers where about to choose between the 2^{32}representation and the 2^{24}one. For the GPU used in the experiment (NVIDIA GT200) both options had advantages and drawbacks. The 2^{32}representation which equals the native word size would allow them to use the available native carry handling instructions for subtraction and addition but required a waste in the cycles by ignoring that the native integer multiplication of this specific GPU is 24-bit. On the other hand, the base 2^{24} representation would avoid this waste but the unused 8 bits per word could not be overlooked. Taking these characteristics into account they decided that the computation waste caused by the base 2^{32}representation was less "harmful" for performance than the storage waste of the base 2^{24}representation so they chose that one. The original Montgomery algorithm requires enough storage in order to complete a full 512-bit multiplication of short-term memory. This is not at all the best possible. A step towards this direction was done by Koç and his team back in 1996 by introducing a method for completing the Montgomery multiplication requiring only *n+3* words of temporary storage. They achieved this by interleaving multiplication with reduction and that way they were able to both reduce the storage needed for processing the words and speed up the modular multiplication. So let us begin with this implementation of the algorithms, the CIOS we previously mentioned. This is the most commonly used approach in software and GPUs. For each word of the modulus, CIOS[13] executes two independent loops: the one for the multiplication process and the other for the reduction one. In each part, the product is multiplied by one digit of the multiplicand, and after that reduced very fast, because the partial product is no more than 32-bit larger than modulus. The drawback of the method is that CIOS generates long carry lines all over the inner loops that make instruction-level parallelism really difficult to achieve. The FIOS is a method[13] that provides a more efficient performance. That is why both multiplication and reduction are included into the inner loop and this loop is iterated $s^2 - s$ times. The great effect of this algorithm in performance derives from the fact that overhead and code size in the outer loop are significantly reduced. The architecture of the processors that this implementation is most suitable for is the RISC one. The only drawback is that it requires two-word quantities to be added, something that involves the possible spreading of a carry over to a third word. Instead, we follow a slightly different approach according to which the need for fast add-with-carry instructions with a redundant representation is alleviated, where two w-bit words is in fact a (w+1)-bit quantity. The last approach, the Montgomery by FIPS, creates the most suitable circumstances for high-leveled parallelism. The outer loop of the FIPS[13] accesses the words of the final product itself and consequently each word of the product can be calculated individually. This method has the disadvantage that it requires more add-with-carry instructions, i.e. a third word to house the resulting carries. Compared to FIOS, FIPS' code has 2 outer loops, and each one of them an inner loop that cannot be unrolled because of the dependency with the outer one. Despite this unrolling problem, the outer loop has a manner of iteration that provides more implicit parallelism and is possible to take advantage of fully unrolled implementations. It also performs some varying memory access patterns which hinders the execution time.

Apart from speeding up the modular multiplication process, the performance can be even better by reducing the number of multiplications required. It was shown that the sliding window method[11] provided the least multiplications. As regards the CUDA implementation, Neves and Araujo assigned each thread to one entire 512-bit exponentiation. In order to maximize the number of the available registers for each thread, they initially set the thread block size to 128, something that ultimately led to poor GPU occupancy. Then, as the recommended thread size was 192, they used this one and were "equipped" with 84 available registers per thread. They did a few more testings and they ended up decreasing the sliding-window size to 4 and increasing the thread block size to 224 threads with 64 threads available then as this combination provided even better performance. Algorithm's code alterations could lead to even faster execution. Loop unrolling as we already mentioned is a technique that always is worth a try and in this case could have a double effect: avoiding the words index accesses by storing them in registers and removing the overhead that caused from the loop control flow. They did these changes manually using PTX assembly tool in the case of performing 32x32-bit wide multiplications and using add-with-carry instructions which are not available in the C language for CUDA. One of the most effective uses of the inline PTX assembly in the implementation was the $a \times b + c$ operation having 32-bit data as inputs and 64-bit as output. It is noteworthy that FIPS was less loop unrolling friendly than FIOS because the former required both inner and outer loop unrolling causing throughput penalty while the latter was loop-friendlier because of the simple single inner loop it contains. The performance results after all these techniques are shown in Figure 4.4, in which we can clearly see that the FIOS approach is the most effective of all. The performance measurements were taken on a system with an NVIDIA GTX260 GPU.

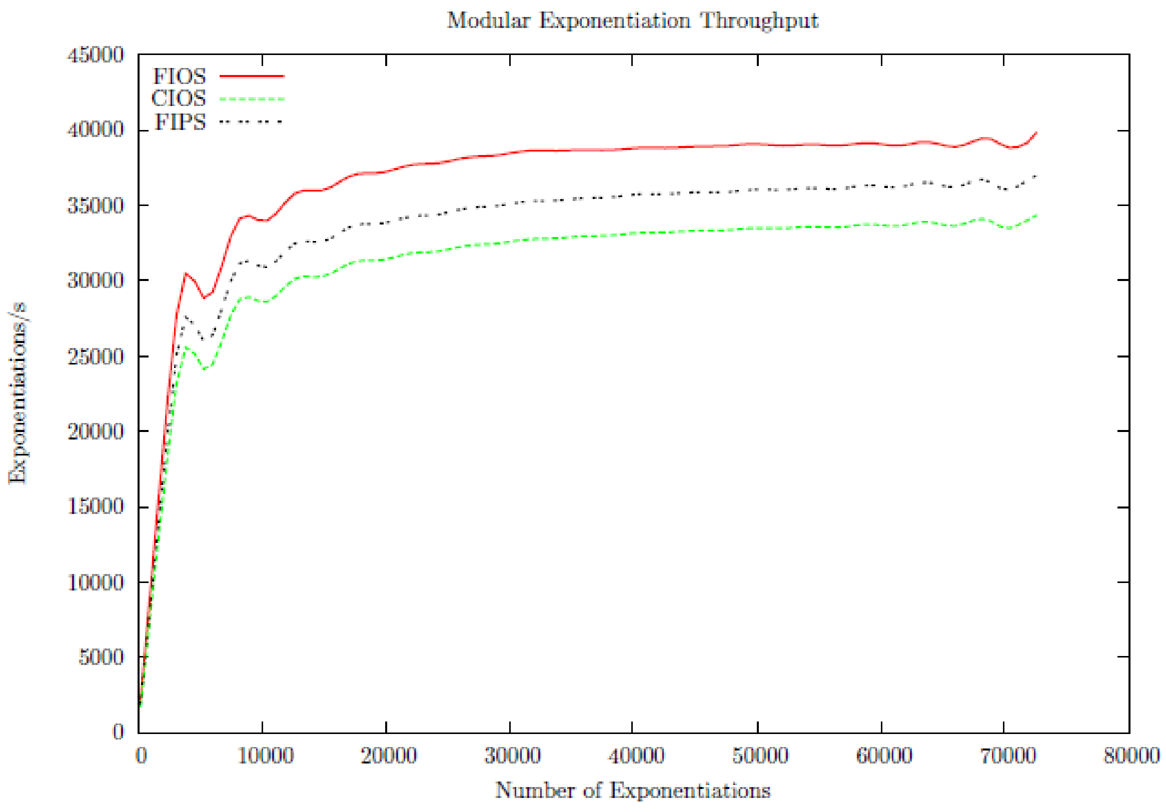


Figure 4.4: GPU throughputs of 512-bit modular exponentiation with different Montgomery approaches[13]

4.1.4 The Chinese Remainder Theorem in RSA decryption process

In the previous section we introduced the Chinese Remainder Theorem (CRT) so in this section we are going to take a closer look at how it is designed and implemented on the RSA decryption process from the perspective of Younis et al.[15]. The RSA decryption process complexity strongly depends on the size of the decryption exponent *d* and the modulus *n*. RSA-CRT is different from the classic RSA implementation in the decryption and the key generation procedures. The decryption procedure is as follows:

Supposing *p* and *q* two numbers coprime positive integers such that *GCD (p, q) ≡ 1.*

If *a ≡ b (mod p)* and *a ≡ b (mod q)*, then *a ≡ b (mod p.q)*

The recipient knows the primes *p* and *q* so the modular operations below can be done by him:

$d_p \equiv d \, mod \, (p - 1)$ and $d_q \equiv d \, mod \, (q - 1)$

$C_p \equiv C \, mod \, p$ and $C_q \equiv C \, mod \, q$

$M_p \equiv C_p d_p \, mod \, p$ and $M_q \equiv C_q d_q \, mod \, q$

Output: $M = [((M_q + q - M_p).\ A) \, mod \, q].p + M_p$

//A ⟵ multiplicative inverse of q determined by the Euclid's extended //algorithm

This implementation results in approximately four times faster execution time since modulus is reduced to half the bit-size of the modulus n meaning that for the computations smaller numbers are used. *p* and *q* are only known to the receiver, so the CRT decryption can only be exploited by him in order to decrypt a message.

The experiment was conducted on a platform its specifications are shown in Table 4.3. At first they implemented the sequential version of the RSA algorithm on the CPU with different key sizes and then recorded the performance and the actual throughput. Secondly, they executed a parallel version of RSA cipher for multi-core GPU and CPU, and measured the results again observing the speed up from the CRT implementation for the decoding process in all these cases. The experiment was separated into two test groups. The first test group consists of varying size messages convenient with the size of the encryption key and specifically one byte less than the size of the modulus. The second group contains a fixed number of messages (600) in order to evaluate the throughput. Again the message size is one byte less than the size of the modulus. The results showed that decryption time of a message is more than the encryption due to the public exponent *(e)* which is smaller than the private exponent *(d)* as we have already analyzed. CRT is used to speed up the decoding process because of the less mathematical calculation, parallelism capability, and for that reason more free resources are available.

Table 4.3. Specification of experiment computer

CPU	Core i7-2670QM (2.20GHz)
CPU Cores (Logical)	8
Memory	12GB
OS	Windows 7 64-bit
HDD	750GB
Processor Cores	96

Table 4.4 depicts the performance gain with CRT for the first test group. The speed up metric is used for that reason with S_1 referring to the speed up for multi-core CPUs and S_2 for the GPUs implementation. For decryption they depict an additional speedup factor S_0 used for better comparison and referring to the original sequential decryption procedure of course without using the CRT.

Table 4.4. Speed up factor for test group 1[15]

Key Size in bits	Encryption		Decryption Without CRT		Decryption With CRT		
	S_1	S_2	S_1	S_2	S_0	S_1	S_2
768	0.644	0.519	0.921	2.079	1.756	2.057	10.121
1024	0.833	0.664	1.160	4.035	2.647	3.351	20.105
2048	0.369	0.547	2.513	9.158	6.158	15.232	118.784
3072	0.683	1.15	3.133	12.06	3.288	11.32	52.817
4096	1.093	1.466	3.164	17.244	4.694	28.368	121.315
6144	1.947	2.874	4.689	20.968	6.695	50.069	425.74
8192	2.255	3.896	6.044	22.998	14.113	118.711	433.004

From the table above we infer that the performance enhancement is linearly increased along with the key length. For encryption, the performance gain is not significant and in fact for key sizes less than 3072 bits the sequential RSA version should be preferred. In the without-CRT case the GPU implementation seems to be approximately 23 times faster even for the small keys. Finally, for the decryption process using the theorem we are most interested in in this experiment, sequential implementation is approximately 14 times faster, the CPU 119 and the GPU ~433 times faster so it is enhanced for all three implementations. The multi-core and many-core units' implementation provides excellent performance because of the composition of the parallelism of the processing and the CRT. As for the throughput evaluation from the test group 2, it decreases as the key length increases which is caused by extensive computation having to do with the higher modulus. In general, decryption using CRT always gives a much higher throughput so from all these results we conclude that the Chinese Remainder Theorem for RSA decryption's modular operations should definitely be preferred whenever possible.

4.1.5 The Pollard P-1 factoring - An RSA optimization over SSL/TLS protocol

We have already mentioned that RSA is a cipher that is used as supplementary to symmetric-key ciphers for secure key exchange or stand alone in digital certificates. Transactions and e-commerce are some of the critical and widely used sections in people's online activity. The secure exchange of data via the net is guaranteed through the Secure Sockets Layer (SSL) and the Transport Layer Security (TLS) which both ensure users with privacy, integrity, and authentication. However, performance is a significant issue every time this procedure is implemented. Research has shown that when the protocol is implemented, 70% of the time spent in the "handshake" stage establishing a secure connection, derives from the RSA implementation. Hence, as we would expect from the topics we have included in this thesis, a parallelized version of RSA would significantly make the SSL/TLS protocol “lighter”. An interesting proposition was made by Vargas et al.[16].

They target the Pollard *p-1* factoring which is a technique developed by Pollard back in 1974 and has its basis on the small Fermat’s theorem and is a method for factoring an integer into its two factors. The method’s steps are the following[17]:

1. Let n be an odd integer that we want it to be factored.
2. Let a = 2 and i = 2.
3. Compute $a = a^{i} \, mod \, n$
4. Compute $d = gcd(a - 1, \, n)$.
5. If *1 < d < n*, then d is a factor of n.
6. If *d = 1*, then set *i = i + 1*, and go back to **step 3**.

In the parallelized algorithm of the Pollard factoring[16] the first thing that happens is the load of the table of the prime numbers to the CPU main memory. Then is transferred to the GPU and stored in the global memory of the unit so the Kernel is ready to be launched. Variable *a* is randomly selected. Once a thread acquires the results of *p* and *q*, it sends them to the GPU memory and then to the CPU one.
For $gridDim.x \, * \, blockDim.x$threads to be executed, in the case you need *Y* times to perform the computation for the whole prime numbers table, in the worst case you have acquired:

$$y = \frac{k}{gridDim.x \times blockDim.x} = \frac{\pi(B)}{gridDim.x \times blockDim.x} =$$

$$\frac{B}{lnB \times gridDim.x \times blockDim.x}$$

The algorithmic complexity of the parallelized Pollard's P-1 algorithm is:

$$O(s \times \frac{B}{lnB \times gridDim.x \times blockDim.x})$$

The implementation was tested on a Linux OS for various combinations of the number of threads according to the RSA key size implementation with the number of threads per black fixed to 64. In particular the tests among with the results are shown below:[16]

RSA Implementations:	CPU Time	GPU Time
● RSA-256 → 4 threads, 64 threads/block	12.47	5.13
● RSA-512 → 8 threads, 64 threads/block	18.93	4.74
● RSA-1024 → 16 threads, 64 threads/block	22.34	5.32
● RSA-2048 → 32 threads, 64 threads/block	28.76	6.12
● RSA-4096 → 64 threads, 64 threads/block	32.87	4.86
● RSA-8192 → 128 threads, 64 threads/block	34.93	4.16
● RSA-16392 → 256 threads, 64 threads/block		

Comparing these results to previous work with the same experiments using Montgomery multiplication for the case of 512-bit and 1024-bit seems to be slower than this. It would not be safe to suppose then that Montgomery is definitely a better option because the experiments were conducted in different architectures. The only way to confirm that is by conducting the experiment in the same system.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this diploma thesis we presented some notable research on how to enhance the performance of symmetric-key block ciphers and the RSA cipher, probably the most important public-key cipher exploiting the modern GPUs. The results were extremely encouraging, proving that there is a variety of techniques that can be exploited in order to make ciphers implementation much faster. In order to decide which of these methods will be more effective, there are few parameters that should be taken into account because for example different architectures would benefit not in the same level from the various implementations. The target of the researchers attempts were mainly the throughput gain achieved by taking advantage of the variety of the CUDA tools provided and also the speed up in the computations which are performed when a cipher is implemented by some parallel programming manual alterations on the code of the ciphers too. Surely, as the GPGPUs capability increases with the time we can definitely expect even more fast cryptographic implementations but of course more experiments with the present technology should be conducted because there is definitely room for even faster implementations as the variety of different systems architectures combined with the enormous computing capabilities and programming tools for GPUs create an almost infinite space for investigation.

REFERENCES

[1] D.Le, “Towards Microarchitectural Design of Nvidia GPUs — [Part 1]”, Apr. 18,2020. [Online].Available:https://medium.com/distributed-knowledge/towards-microarchitectural-design-of-nvidia-gpus-part-1-abe2fd5d9e52 [Accessed: Sept. 17, 2021].

[2] Q. Li, C. Zhong, K. Zhao, X. Mei and X. Chu, "Implementation and Analysis of AES Encryption on GPU," *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012, pp. 843-848, doi: 10.1109/HPCC.2012.119.

[3] C. Paar and J. Pelzl, *Understanding Cryptography - A Textbook for Students and Practitioners.* Springer, pp.89-90, 115, 124-132, 2010.

[4] O. Harrison and J. Waldron, “Practical Symmetric Key Cryptography on Modern Graphics Hardware.,” in *USENIX Security Symposium*, pp. 195–210, 2008.

[5] K. Iwai, T. Kurokawa, and N. Nishikawa, “AES Encryption Implementation on CUDA GPU and Its Analysis.,” in *ICNC*, pp. 209–214, 2010.

[6] K.Aoki et al., ”Specification of Camellia-a 128-bit Block Cipher”, Sept. 26, 2001[Online].Available:https://www.cryptrec.go.jp/en/cryptrec_03_spec_cypherlist_files/PDF/06_01espec.pdf [Accessed: Sept. 17, 2021].

[7] W.-K. Lee, B.-M. Goi, R. C.-W. Phan, and G. S. Poh, “High speed implementation of symmetric block cipher on GPU.,” in *ISPACS*, pp. 102–107, 2014.

[8] N.Hoffman, “A simplified IDEA algorithm”,March 2007.[Online].Available:https://www.nku.edu/~christensen/simplified%20IDEA%20algorithm.pdf [Accessed: Sept. 17, 2021].

[9] B. Schneier, ”The Blowfish Encryption Algorithm”. [Online].Available: https://www.schneier.com/academic/blowfish[Accessed: Sept. 17, 2021].

[10] J. Gilger, J. Barnickel, and U. Meyer, “GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library.,” in *ISC*, vol. 7483, pp. 338–353, 2012.

[11] S. Mahajan and M. Singh, “Analysis of RSA algorithm using GPU programming.,” *CoRR*, vol. abs/1407.1465, 2014.

[12] H. Fadhil and M. I. Younis, “Parallelizing RSA Algorithm on Multicore CPU and GPU.,” in *International Journal of Computer Applications,* vol. 87, no.6, pp. 15-22, Febr. 2014. doi: 10.5120/15211-3704.

[13] S. Neves and F. Araújo, “On the performance of GPU public-key cryptography.,” in *ASAP*, pp. 133–140, 2011.

[14] https://crypto.stanford.edu/pbc/notes/numbertheory/crt.html[Accessed: Sept. 17, 2021].

[15] M.I. Younis, H. Fadhil, Heba & Z. Jawad, (2016). “Acceleration of the RSA Processes based on Parallel Decomposition and Chinese Remainder Theorem”., in *International Journal of Application or Innovation in Engineering & Management*., vol. 5, no.1, pp. 12-23, Jan. 2016.

[16] M. P. Pineda Vargas, R. A. A. Rodriguez and O. J. Salcedo Parra, "Algorithm for the Optimization of RSA Based on Parallelization over GPU SSL/TLS Protocol," 2017 IEEE International Conference on Smart Cloud (SmartCloud), pp. 294-297, 2017. doi: 10.1109/SmartCloud.2017.55.

[17] M.S. Lydia, M.A. Budiman, Mohammad and D. Rachmawat, “On using Pollard’s p-1 Algorithm to Factor RPrime RSA Modulus,.” *International Conference of Science, Technology, Engineering, Environmental and Ramification Researches,* Jan. 2018. doi:1895-1899 10.5220/0010083618951899.