TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Acceleration of the Adiantum Algorithm on Embedded GPUs

Author:
Polykratis GEORGIS

Thesis Committee:
Assoc. Prof. Sotirios IOANNIDIS
Prof. Apostolos DOLLAS
Assoc. Prof. Vasilis SAMOLADAS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

July 6, 2022

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Acceleration of the Adiantum Algorithm on Embedded GPUs

by Polykratis GEORGIS

In the age of information the need for security becomes more and more obvious. Cryptography is a powerful tool providing the necessary confidentiality on sensitive data. There are many cryptography algorithms, each with its own strengths and weaknesses that constantly compete against numerous threats and of course, time. Adiantum is a lightweight alternative providing, for the time being, both speed and security. This thesis focuses on speeding up of this algorithm on large plaintexts using the parallelism offered by a modern GPU. It consists of the detailed profiling and analysis of the algorithm followed by the implementation of its slowest component, XChacha Stream cipher, on Jetson Xavier NX developer kit. Results suggest a x6 speedup on Adiantum_XChacha20, x4 on Adiantum_XChacha12 and x3 Adiantum_XChacha8 with a x4.5, x3.5 and x2.5 energy reduction on the three versions respectively. These results lead to the conclusion that, using the GPU, it is possible to upgrade security by using Chacha20 without compromising neither, speed or energy consumption.

Acknowledgements

I would like to express my deepest gratitude to my advisor Dr. Dimitrios Theodoropoulos for his help and guidance throughout this thesis. His continuous support and diligence were crucial for the completion of this project even through a tightly scheduled year for both of us.

I would also like to thank my supervisor, Prof. Sotirios Ioannidis for teaching me about security and specifically the field of cryptography and making the understanding of the Adiantum cryptography algorithm a lot easier.

In addition, i would like to thank my thesis committee Prof. Vasilios Samoladas and Prof. Apostolos Dollas that have been a great inspiration to me throughout my studies and helped a lot in choosing this field of research, for evaluating my work.

Special thanks to Dimitris Deyannis for his guidance and help in understanding embedded GPUs. He has been a most valuable guide through this new field.

Last but not least i would like to express how grateful i am to all my friends and family for standing by my side at all times even when neglected in my tight7y 7 schedule. Finally, i have to thank my roommate Luna for keeping up with me through the last two years and for letting me stay at her house. Us living together has been a whole new experience continuously broadening my horizons.

Contents

Abstract iii

Acknowledgements v

Contents vii

List of Figures ix

List of Tables xi

List of Algorithms xi

List of Abbreviations xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Scientific Contributions . . . 2
- 1.3 Thesis Outline . . . 3

2 Related Work 5
- 2.1 Hardware Design of ChaCha20 and Poly1305 . . . 5
- 2.2 Hardware Acceleration of Adiantum Cryptography Algorithm on PYNQ . . . 6
- 2.3 CryptoGraphics: Secret Key Cryptography Using Graphics Cards 6
- 2.4 Cryptographic algorithm acceleration using CUDA enabled GPUs . . . 7
- 2.5 A hybrid CPU/GPU Scheme for Optimizing ChaCha20 Stream Cipher . . . 8
- 2.6 Thesis Approach . . . 8

3 Architecture Analysis 9
- 3.1 Fundamentals of cryptography . . . 9
- 3.2 Adiantum . . . 10

3.3 Profiling . . . 12
3.4 Software Analysis and Optimization . . . 13
3.4.1 XChacha . . . 14
3.4.2 Endianness . . . 17
3.4.3 Software optimization of Chacha . . . 18

4 GPU Implementation **21**
4.1 Tools Used . . . 21
4.1.1 CUDA-GDB . . . 21
4.2 GPU Platform . . . 21
4.2.1 JETSON XAVIER NX specifications . . . 22
4.3 GPU distinctive features . . . 24
4.3.1 The GPU Perspective . . . 24
4.3.2 SIMT Approach . . . 24
4.3.3 GPU memory . . . 25
4.4 CUDA implementation of Chacha . . . 26
4.4.1 Chacha paralellization . . . 27
4.4.2 CUDA kernel . . . 29
4.4.3 Kernel wrapper . . . 31
4.4.4 Linking issues . . . 32

5 Results **33**
5.1 Specification of Compared Platforms . . . 33
5.1.1 Initialization issue . . . 33
5.2 Throughput and Latency Speedup . . . 34
5.3 Adiantum Performance . . . 34

6 Conclusions and Future Work **43**
6.1 Conclusions . . . 43
6.2 Future Work . . . 44

References **47**

List of Figures

3.1 Adiantum algorithm overview 11
3.2 Chacha initial state matrix 14
3.3 Columns and diagonals of Chacha state matrix 15
3.4 Standard Chacha halfround routine 16
3.5 Example of endianness 17
3.6 Simplified computer memory hierarchy 18

4.1 Jetson Chip 22
4.2 Jetson Specifications 23
4.3 Jetson Xavier cache coherence 26
4.4 Flowchart of Chacha_generic 27
4.5 Flowchart of the GPU implementation of Chacha 28
4.6 Kernel code of Chacha_generic 29
4.7 Kernel wrapper code 31

5.1 Performance and speedup results 36
5.2 Point of acceleration of Adiantum_XChacha on the GPU . . . 38
5.3 Adiantum throughput comparison 40

List of Tables

3.1 XChacha time percentage from Adiantum profiling 13

5.1 XChacha20 encryption energy consumption for all 4 variations 41
5.2 XChacha12 encryption energy consumption for all 4 variations 41
5.3 XChacha8 encryption energy consumption for all 4 variations 41

List of Abbreviations

AES	**A**dvanced **E**ncryption **S**tandard
ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
GDB	**G**NU **D**e**B**ugger
GPIO	**G**eneral **P**urpose **I**nput **O**utput
GPU	**G**raphic **P**rocessor **U**nit
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
LUT	**L**ook **U**p **T**able
MAC	**M**essage **A**uthentication **C**ode
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
NVDLA	**NV**IDIA **D**eep **L**earning **A**ccelerator
OCB	**O**perational **C**ipher **B**lock
PCIe	**P**eripheral **C**omponent **I**nterconnect **E**xpress
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SHA	**S**ecure **H**ash **A**lgorithm
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SIMT	**S**ingle **I**nstruction **M**ultiple **T**hread
SoC	**S**ystem **o**n **C**hip
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
USD	**U**nited **S**tates **D**ollar

Dedicated to my family and friends…

Chapter 1

Introduction

Technology has become an integral part of scientific research and innovation, as well as an inextricable part of our everyday lives. Computer science has evolved above and beyond the scope imaginable two decades ago and continues to evolve exponentially even after the rejection of Moore's law [1].

With every passing day, information becomes more and more valuable, to the point that entire nations are spending enormous amounts of money for the acquisition of confidential information. Even in a smaller scale, personal information is consistently used as a product for advertising companies. The constant information war has made the need for security indisputable.

One major aspect of computer security is cryptography. Cryptography has become a necessity for both storing and transmitting information, but as with every computing procedure, it takes time and "time is money" says the famous idiom written by Benjamin Franklin. As such, it becomes obvious that making cryptographic algorithms both faster and more secure is a meaningful research subject. For the time being, better security is achieved by combining disk and file encryption, whereas acceleration can be achieved by using hardware benefits not yet present in software applications.

1.1 Motivation

The word cryptography comes from the Greek words "krypto" which means "to hide" and "grapho" which means "to write". People have been using it to hide information for thousands of years, mostly to keep military plans from being compromised. Of course in contrast to these ages, modern cryptography is impossible to be ciphered or deciphered by hand due to its huge complexity. Actually it was during the second world war when the "father of

computer science", Alan Turing, created the first premature version of a computer [2]. Its purpose to decrypt the messages encrypted by the "Enigma", the German cipher. So it can be stated that computers and cryptography are deeply linked.

In computer science, cryptography is the procedure where a readable chunk of data called plaintext in converted to an unreadable one called ciphertext and vice versa. The two procedures are called encryption and decryption respectively. Symmetric-key ciphers use the same secret key for encryption and decryption, whereas asymetric-key ciphers use a public key for encryption and a secret one for decryption [3].

Cryptographic algorithms are hard to implement so as to be secure enough and at the same time fast enough to be useful. As mentioned, the advantages provided by different processing systems can be exploited for the acceleration of such algorithms to offload the CPU. Those alternative processing systems are presented below.

ASICs are processing units designed to run optimally the specific function they were programmed for. Combining low power consumption and high parallelism they are a great choice for running cryptographic algorithms in the background. Unfortunately their lack of versatility and high production cost make them a less feasible solution.

FPGAs combine the versatility of a CPU with the high parallelism of an ASIC. With a reasonable power consumption they are a powerful processing platform at the hands of an experienced developer because they have huge capabilities for optimizations on both software and hardware aspects of a program.

GPUs offer the greatest amount of parallelism in modern hardware design choices. With a relatively high power consumption they are widely used for display purposes. They are the fastest choice for applications with very high parallelism capabilities.

1.2 Scientific Contributions

The goal of this thesis was to expand upon Kostantinos Ampatzidis work [4] and further improve Adiantum performance on large plaintexts, using the state of the art embedded GPU provided by NVIDIA in Jetson Xavier NX. Adiantum was chosen because it is a relatively new encryption algorithm

that combines many fundamental cryptography models, making it a compelling research subject. Because of its age Adiantum has much room for improvement both in speed and in security. Our approach, focusing on the acceleration of the algorithm can be summed up with:

- Profiling of the algorithm to determine its most time consuming parts.
- The acceleration of those parts in the GPU using the multi thread benefit.

Adiantum is the cryptography algorithm that this thesis is centered around. It is a relatively fast cryptography algorithm developed by Google in December 2018, originally for disk encryption [5]. Because of its ability to handle varied message lengths, making it versatile, it is an interesting choice fro optimization through the parallelization capabilities provided by the Jetson Xavier NX GPU.

Our results proved very encouraging achieving about x6 times speedup for the Adiantum_XChacha20 variation on large messages which is the slowest and most secure version. On Adiantum_XChacha12, an approximate x4 times speedup and on Adiantum_XChacha8 a x3 times speedup using the largest amount of parallelization possible provided by the Jetson GPU. A most interesting conclusion though is the fact that using a multi-thread approach, the same speed is achieved regardless of the number of rounds of Chacha, offering the ability to upgrade security without compromising speed.

1.3 Thesis Outline

- **Chapter 2 - Related Work:** This chapter lists all related work and presenting the tools used to complete this thesis.
- **Chapter 3 - Robustness Analysis:** This chapter presents of the profiling results, along with detailed description of Adiantum cryptography algorithm and its most time consuming part. In addition it introduces a software optimization of the algorithm.
- **Chapter 4 - GPU Implementation:** This chapter analyzes of the implementation of Chacha in the GPU using CUDA, explaining the choices made throughout development.

- **Chapter 5 - Results:** This chapter offers the visualization of the results and their detailed analysis for different implementations of Adiantum.
- **Chapter 6 - Conclusions and Future Work:** This chapter concludes everything that occurred through the results and presents future work that may further optimize the algorithm.

Chapter 2

Related Work

There has been a relevant research on the implementation and acceleration of Adiantum from a hardware, specifically an FPGA, point of view by Kostantinos Abatzidis [4] of Technical University of Crete. This thesis has been inspired by it so as to compare a GPU and a FPGA approach. Furthermore, it is important to present some relevant work on different implementations of Chacha, the part of Adiantum that was the center piece of this thesis. ([6],[7],[8])

2.1 Hardware Design of ChaCha20 and Poly1305

The concept of optimizing Chacha20 and Poly1305 through hardware while maximizing throughput and minimizing area was explored by Guard Kanda and Kwangki Ryoo in their paper "High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses Cha Cha20 and Poly1305" [6]. Chacha20 and Poly1305 are both fundamental parts of the Adiantum algorithm. The paper includes the design of hardware, specifically for these two algorithms, in HDL-Verilog.

The hardware designed for the optimization of Chacha20 consists of the following parts. A Little endian serializer that converts the 64 bytes of the initial state matrix little-endian format. The initial state matrix creator, called Init_State_Matrix, converts the result into the Chacha initial state matrix. The state matrix is put through the Chacha State Generator block. That block consists of the appropriate hardware which is able to perform 4 quarter rounds either in parallel or serially, and adds the result to the original state matrix. The final product is then XORed with the plaintext. The entire procedure in controlled via FSM.

2.2 Hardware Acceleration of Adiantum Cryptography Algorithm on PYNQ

Inspired by the work of G.Kanda and K.Ryoo [6] a thesis on the hardware acceleration of Adiantum, by Kostantinos Abatzidis [4] of Technical University of Crete, has been the main influence for this thesis. In this work, K. Abatzidis profiled the algorithm, finding that XChacha12 stream cipher is the most time consuming part. The cipher was implemented on a Pynq FPGA platform achieving a speedup of up to x10,731, thus achieving Amdahl's law theoretical speedup.

This work uses pipelining to avoid delays from consecutive runs of the 12 rounds of Chacha for every 64 bytes of a large plaintext. At the same time he needed to minimize and improve the data transactions between the PL and PS of the FPGA. This was achieved by taking advantage of the independent parts of the algorithm, the 12 rounds and adding the result to the initial matrix and putting those in the PL pipeline returning the results through the AXI stream to the AXI DMA and finally back to the PS.

The approach used in this work is very similar to our approach with the main differences being:

- The GPU offers complete parallelization instead of a pipeline.
- the Adiantum version used in this thesis is in C and is an optimized version of the python one researched by this work.
- This thesis explores Adiantum_XChacha20 and Adiantum_XChacha8 along with Adiantum_XChacha12.

2.3 CryptoGraphics: Secret Key Cryptography Using Graphics Cards

In this paper Debra L. Cook, John Ioannidis, Angelos D. Keromytis and Jake Luck [9] study the potential of using GPUs on symmetric key ciphers so as to accelerate cryptographic processing and to offload system resources. They research the potential use of graphics cards on both stream and block ciphers and realize the operations that render certain ciphers unsuitable for optimization using GPUs. Several experiments on the potential parallelism of stream ciphers take place, as well as proving that running AES on the GPU

is ill-suited to offload the CPU given current APIs. In addition they study the potential of encrypting images directly into the GPU without their conversion to plaintext into the system memory first.

This work proves that the GPU APIs of the time were unsuitable for certain byte level operations present in many stream and block ciphers. As for direct GPU image encryption, it can be beneficial as long as some issues regarding compression, dithering and safe storage of the secret key. In relation to this thesis, this work uses the GPU in a similar way when experimenting with XORing on stream ciphers. Specifically they determine that operations on small data sizes cannot benefit from the GPUs high parallelism mainly because of data transfers.

2.4 Cryptographic algorithm acceleration using CUDA enabled GPUs

In this thesis, Maksim Bobrov, of the Rochester Institute of Technology [7] tested the acceleration of three different encryption algorithms, AES, SHA-2, and Keccak, using CUDA. Testing was performed on the encryption and hashing algorithms using a single-kernel approach very similar to ours, a multi-kernel approach, and the potential speedup of offloading from the CPU to the GPU.

The thesis proved that speedup through the GPU can be achieved only if the algorithm exhibits enough parallelism. AES gets x2.6 times faster with a single-kernel approach, while SHA-2 and Keccak that do not offer enough parallelism show a performance decrease. The multi-kernel approach proves to be x3.6-4.7 times faster for the AES. Another really important result is the reduction Reduction in CPU time by offloading the encryption algorithms from it, allowing the CPU to perform other tasks. A 40–60% reduction was observed with the GPU implementation of AES, a 22–52% for SHA-2 and approximately 39% for Keccak.

2.5 A hybrid CPU/GPU Scheme for Optimizing ChaCha20 Stream Cipher

In their paprer Ziheng Wang, Heng Chen and Weiling Cai, of the School of Computer Science and Technology, Xi'an Jiaotong University [8], tried different schemes to accelerate Chacha20 algorithm. They used a multi core CPU achieving better performance than that of OpenSSL. Another implementation was using a GPU and accelerating the Chacha block function by exploiting coalesced memory access and inline Parallel Thread Execution, achieving an outstanding peak throughput of 211.41GB/s. Finally they tried a hybrid CPU/GPU implementations achieving a 87.56% peak bidirectional bandwidth of a PCIe channel.

The approach on accelerating Chacha in this paper, although seems similar to ours, it is much different. In the paper, the developers chose ti internally parallelize the Chacha20 block function, making use of various smart schemes to optimize throughput and memory usage. Our approach focuses on executing in parallel as many instances of Chacha20 block function as possible. This approach is called external parallelism. Also our approach on maximizing memory throughput depended on utilizing registers as much as possible minimizing the memory reads and writes.

2.6 Thesis Approach

This thesis aims to explain how the Adiantum algorithm works and how acceleration can be achieved using the high parallelism provided by a modern GPU. The accomplished acceleration is a result of the following processes:

1. Profiling of Adiantum, dictating its slowest parts and determining their ability for parallelism.
    - Section 3.3 Profiling
2. Implementation of Chacha in the GPU using CUDA, taking advantage of the parallelism offered.
    - Section 4.4 CUDA implementation of Chacha.

Chapter 3

Architecture Analysis

3.1 Fundamentals of cryptography

Symmetric cryptography is divided in Stream[10] and Block ciphers[11], whose features will be briefly analyzed below because of their importance in the Adiantum algorithm. Of equal importance are hash functions and MAC both utilized by Adiantum.

Stream ciphers, synchronous or asynchronous, encrypt each bit individually by combining (usually XOR) plaintext bits with key stream bits. In synchronous stream ciphers the key stream is independent from the ciphertext, whereas in asynchronous stream ciphers the key stream utilizes the ciphertext.

Block ciphers, in contrast to stream ciphers, encrypt an entire block of n plaintext bits using the same key of a specific size, to n bit ciphertext [11]. In block ciphers each block is encrypted/decrypted independently from other blocks, a feature really useful for parallelism as will be shown below. AES 256 block cipher of 16 bytes is used in Adiantum with a 256 bit key.

Hash functions, widely popular for data storage applications, are used to map arbitrary sized messages into fixed length bit strings. The return values are called hash values and are used as an index to the fix-sized table created.

MACs are tags used to authenticate messages, checking if they have been altered through transmission, or even if they were sent from the right source. In order to do that, they use a specific key. MACs are symmetric key schemes, and they depend on hash functions or block ciphers. As a result they are much faster than digital signatures.

3.2 Adiantum

Adiantum can be divided into four procedures. A hash function consisting of NH [12] combined with Poly1305 [13] followed by a single AES-256 invocation block cipher. An implementation of XChacha stream cipher and finally, there is the same hash function consisting of NH combined with Poly1305

Adiantum [5] is a variable-input-length [14], tweakable block cipher [15] developed to as a fast and secure encryption algorithm. Each of the previously mentioned procedures needs a different sized key, specifically 16-byte Poly1305 for tweak 16-byte Poly1305 for message 1072-byte NH 32-byte AES and as a result, XChacha is used in the beginning to generate the aforementioned keys within a 1136 byte stream. Further key generation within the algorithm uses those keys for initialization. As for the tweak key, in order to keep it simple it is similar to an initialization vector for a CBC mode (Cipher Block Chaining) or a nonce for OCB mode (Offset Codebook) [16].

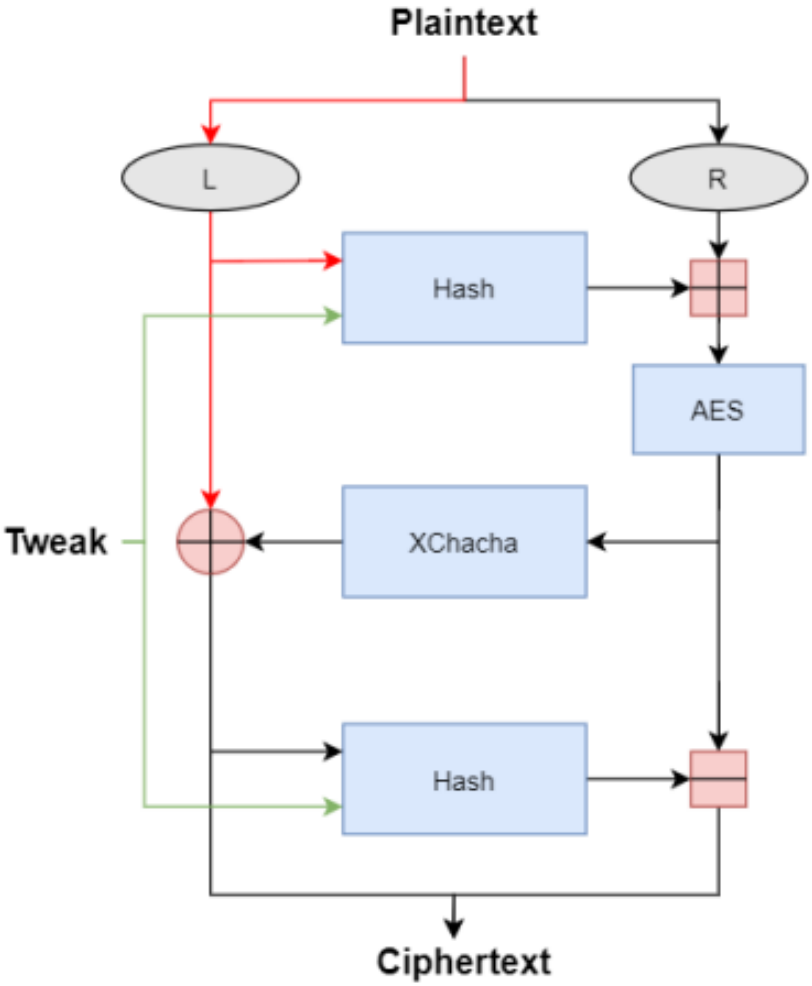

FIGURE 3.1: Adiantum algorithm, courtesy of Kostantinos Ampatzidis in [4]

As shown in figure 3.1 the plaintext is divided in right, consisting of the last 16 bytes of the message, and left consisting of the rest. The left part is put in the hash fuction, the result, along with the right part are encrypted using the AES-256 block cipher followed by XChacha. The created text is then XORed with the original left text and hashed again, resulting in the ciphertext. Decryption algorithm follows the exact same steps with AES-256 and XChacha decryption algorithms.

3.3 Profiling

Adiantum was mainly designed by Google for disc encryption, thus the test vectors provided in the algorithm for testing are up to 4096 bytes of plaintext. Adiantum, though, can be used for messages up to 2^{73} bits with one key. As a result, profiling of the algorithm was essential for this thesis. In order not to misinterpret it as benchmarking, profiling is used to determine the slowest part of the algorithm where optimization, and specifically parallelization is possible. The whole procedure started with deeply understanding the C version of the algorithm provided by Google. The repository contains a python version of the algorithm and an optimized C version, which has benchmarks for a number of different encryption algorithms such as HpolyC and noekeon, used mainly for comparison. The benchmarks were set to test messages greater of equal to 1MB. To make profiling accurate, the algorithm test procedure was removed, along with all other encryption algorithms except from Adiantum-XChacha8, Adiantum-XChacha12 , Adiantum-XChacha20. Very important was the removal of the 1MB minimum plaintext limit so as to test various small messages as well a large ones. It is also important to mention that Google provides most of the aforementioned algorithms with a generic and an SIMD version, specific to different cores such as ARM and x86_64.

Profiling was done on the Jetson CPU 4.2, on a Linux operating system using Gprof, GNU project's profiler ideal for profiling C applications. The procedure starts by running the Adiantum executable, having previously profiling enabled. We tried various message lengths, starting at 128 KB and up to 256 MB. Then the profiler provides files that gave detailed analysis on call count of each function, times it was called, time percentage consumed in each function and what percentage of the whole algorithm execution time was consumed in each function and its subroutines. It also provides the time consumed in each function but its unreliable because the profiling of an algorithm is exponentially slower than a single run.

From [4] it has been proven that encryption and decryption cost about the same in time since they are the exact same procedure in reverse. Table 3.1 shows the time percentage consumed in the slowest function in all three versions of the algorithm. More specifically it is shown that chacha_generic takes about 73.5%of the total time for Adiantum-XChacha8, about 81% of the total time for Adiantum-XChacha12 and about 90% of the total time for Adiantum-XChacha20, percentages that don't drastically change with the

TABLE 3.1: XChacha time percentage from Adiantum profiling

Message(bytes)	XChacha8	XChacha12	XChacha20
128k	40.0%	74.8%	79.3%
256k	70.0%	78.5%	95.1%
512k	64.3%	73.7%	89.3%
1m	75.0%	78.6%	91.9%
2m	72.1%	80.40%	88.3%
4m	78.4%	85.9%	88.1%
8m	74.4%	82.5%	87.4%
16m	71.5%	80.0%	91.1%
32m	75.3%	81.6%	88.9%
64m	81.6%	82.3%	93.1%
128m	83.2%	81.8%	93.5%
256m	75.5%	89.0%	92.5%

message size, although execution time linearly rises with the plaintext length. The deviations observed are most likely due to the different cache miss rate in each run of the algorithm. Even though Adiantum uses two hash functions combined, twice, Chacha stream cipher is by far the most time consuming, even in its fast form, Chacha8. By using Amdahl's law formula 3.1 on the results of the table,

$$MaxTheoreticalSpeedup = \frac{1}{1-p} \tag{3.1}$$

we can deduce that the maximum speedup that can be achieved by infinitely speeding up Chacha is about 3.8 times for Chacha8, about 5.25 times for Chacha12 and about 10 times for Chacha20.

3.4 Software Analysis and Optimization

From the profiling of the algorithm, it is evident, that XChacha stream cipher takes up most of the execution time. As a result it is important for the purposes of this thesis, to analyze in depth XChacha stream cipher, which will be the main focus of the optimization process. Each version of Adiantum uses respectively XChacha8, XChacha12 and XChacha20. XChacha [17] comes from Chacha [18] which is a variation of Salsa Stream Cipher [19]. Chacha is an optimized version of the Salsa algorithm, all three Chacha versions (8/12/20) work very similarly and the number refers to the number

of rounds performed by the cipher. The only difference is that more rounds equal better security and more execution time.

3.4.1 XChacha

XChacha has two steps:

- A single HChacha iteration.
- Followed by the original Chacha algorithm.

It is important to note here that both Chacha and HChacha have a generic and an SIMD version and this thesis will be focusing on the generic version.

Chacha initial state as shown in figure 3.2:

- 16-byte constant “expand 32-byte k”
- 32-byte key
- 16-byte initialization vector, whose first 4 bytes are also used as a counter.

Cosnstant "expa"	Cosnstant "nd 3"	Cosnstant "2-by"	Cosnstant "te k"
key[0]	key[1]	key[2]	key[3]
key[4]	key[5]	key[6]	key[7]
Counter/ iv	iv	iv	iv

FIGURE 3.2: Chacha initial state matrix

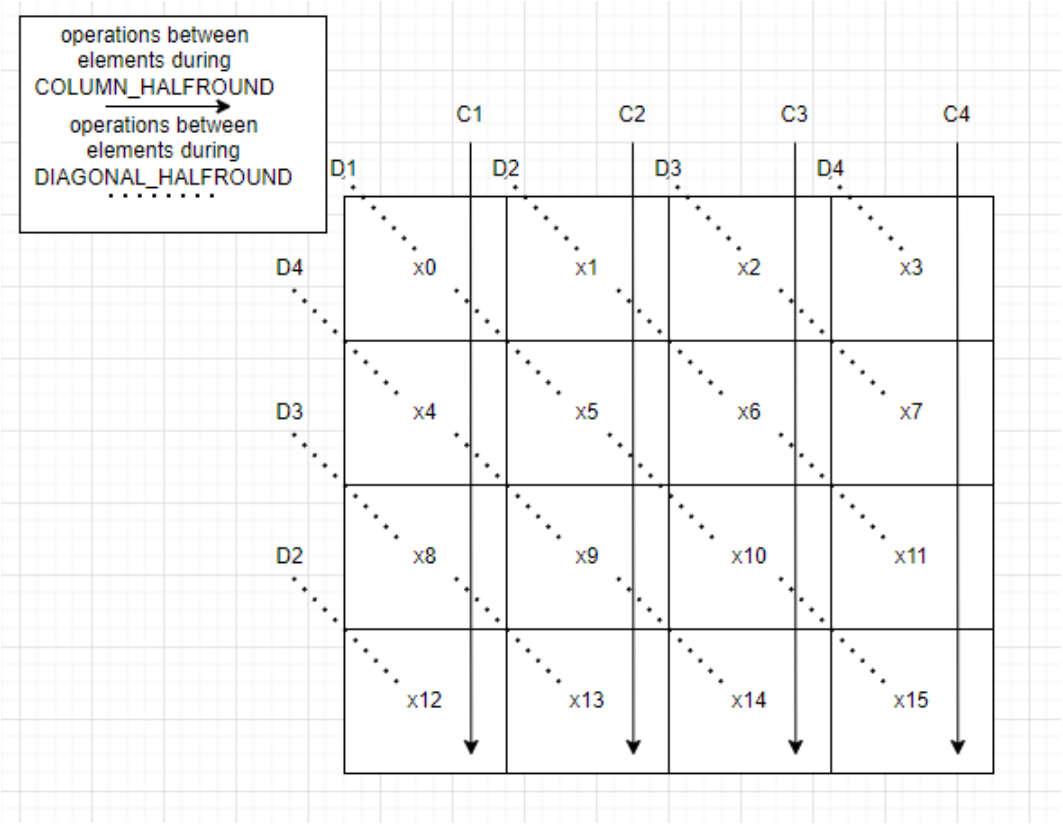

FIGURE 3.3: Columns and Diagonals of the Chacha state matrix's copy: x

The procedure can be analyzed in the following steps. During the first step, a copy of the Chacha initial state, undergoes a Chacha number of rounds divided by two ([8/12/20]/2) loop, called chacha_perm_generic consisting of two column half-rounds followed by two diagonal half-rounds. As a result, each loop iteration represents a Double-Round. During each Double-Round each column undergoes twice the procedure shown demonstrated in figure 3.4. Then the exact same action is taken twice for each diagonal. More specifically each operation between elements in each diagonal half-round is done between consecutive elements in the same diagonal (figure 3.3). For example, adding line 1 (elements 0,1,2,3) to line 2 (elements 4,5,6,7) in the diagonal half-round corresponds to adding elements 0 and 5, 1 and 6, 2 and 7, 3 and 4.

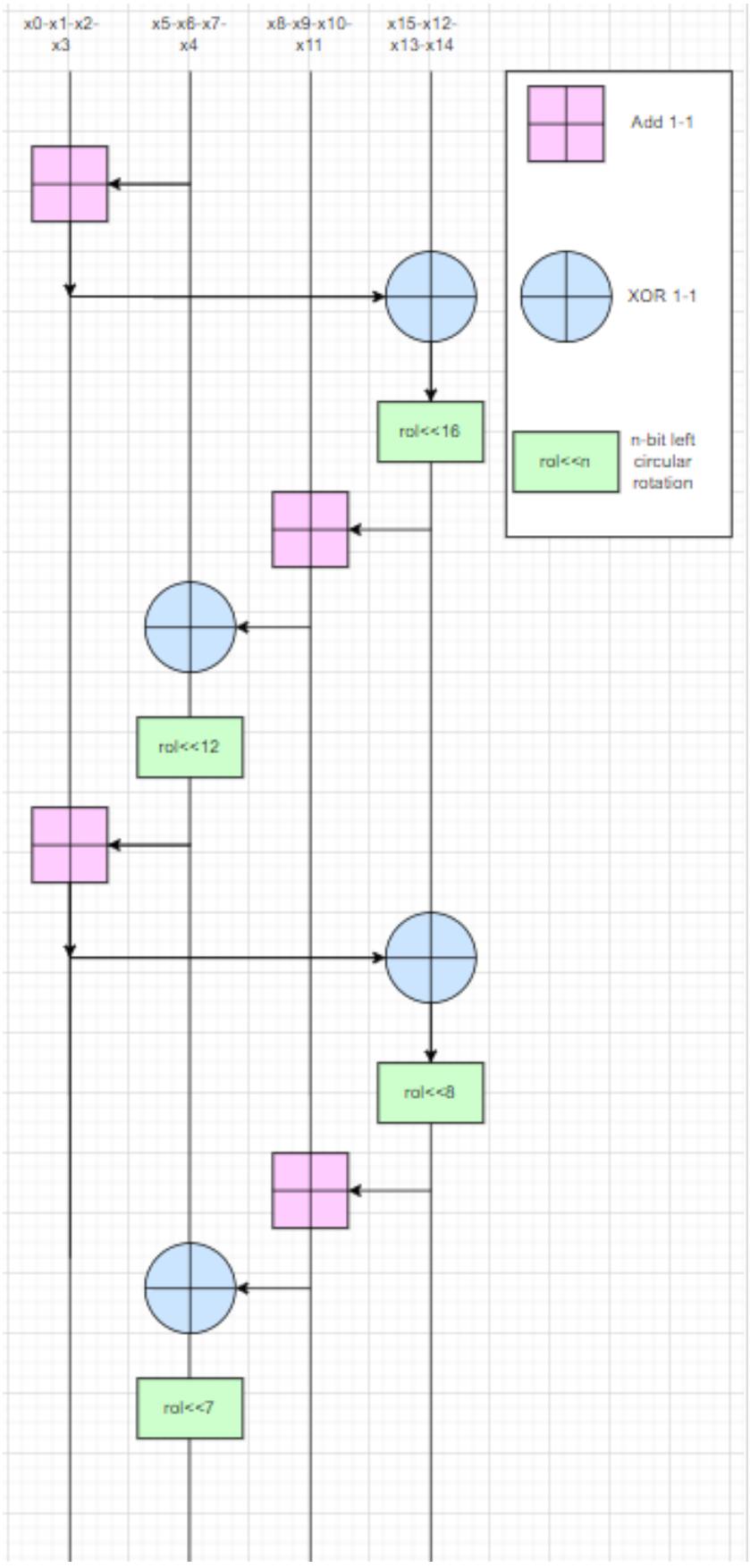

FIGURE 3.4: Standard Chacha halfround procedure for either column or diagonal

After all Double-Rounds have finished, the result matrix is added to the original state matrix forming the stream, whereas the counter is increased. Finally the XOR operation is used between the stream and the message. The aforementioned procedure encrypts one 64-byte Chacha block and has to be repeated for the entirety of the plaintext.

HChacha and XChacha: HChacha uses the same initial table as Chacha and goes through the standard chacha_perm_generic, saving the first and last 16 bytes of the result matrix, both in little endian format and are combined to form the XChacha subkey. XChacha constructed by the HChacha intermediate state followed by the classic Chacha algorithm. HChacha is used to create the subkey used in Chacha.

3.4.2 Endianness

In computing, the method of storing data is called endianness and is distinguished in two major categories "Little Endian" and "Big Endian" [20]. To simplify, big endian means that the computer reads memory from left to right whereas little endian means the opposite. When a machine stores data in big endian order, the most significant bytes are stored in the first memory location and all the other bytes follow. When a machine stores data in little endian order instead, the least significant bytes are stored in the first memory location, and the other bytes follow accordingly, up to the most significant bytes that are stored in the last one. It is important to mention that, despite the endianness of bytes in a computer, bits are always stored in big endian order.

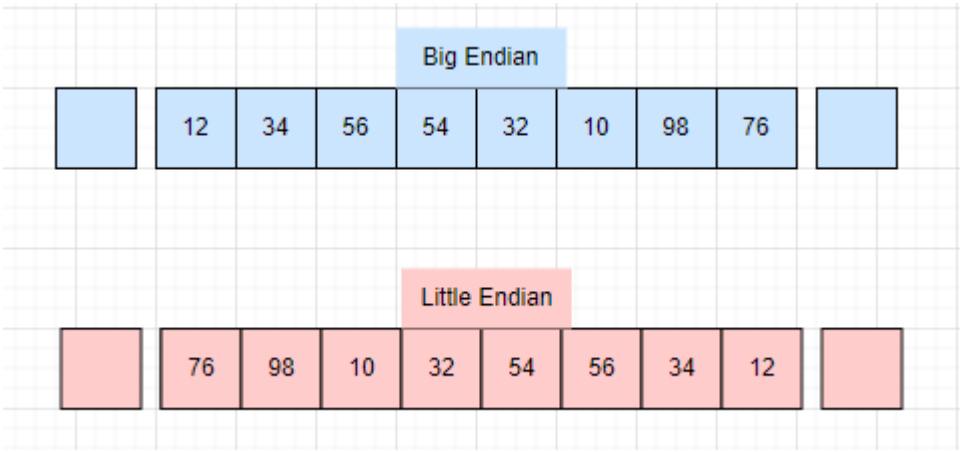

FIGURE 3.5: Endianness toy example

Little endian order has a major speed advantage over big endian order and that is that least significant bytes stay intact as more digits are added to higher addresses, whereas they would have to change positions if they were in big endian order. Assuming that each integer is stored as 4 bytes, then a toy example of a variable 0x1234565432109876 is in figure 3.5 where the first memory position is the first block on the left.

3.4.3 Software optimization of Chacha

While the acceleration of Chacha algorithm from the GPU perspective will be analyzed in the next sections, it is important to present some software optimizations that proved to greatly improve the performance of the generic version of Adiantum. Profiling of Chacha has proved that the most time consuming part of the algorithm is the chacha_perm_generic function described in 3.4.1 mainly because of the 32 bit left rotation. Unfortunately this function cannot be accelerated using the advantages of the GPU because of the several data transfers needed. Another reason for the slow speeds of this specific function are the numerous consecutive memory reads and writes of the state matrix copy, called the x matrix.

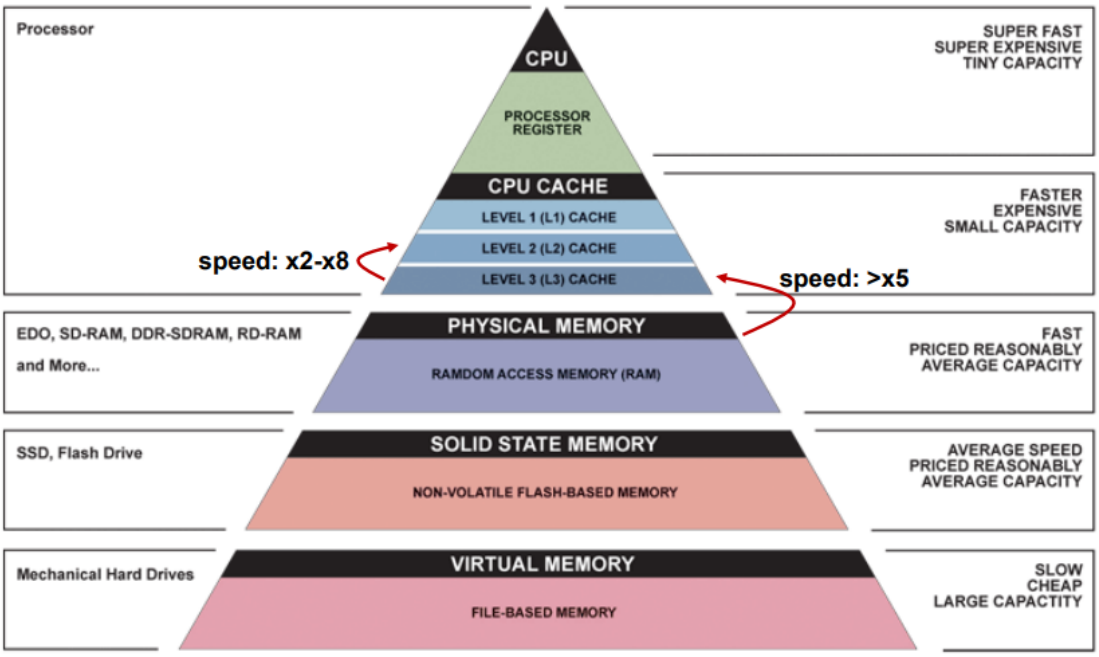

FIGURE 3.6: Simplified computer memory hierarchy, courtesy of Ryan J. Leng [21]

In computing accessing data from the L1 cache is always slower than reading/writing data to registers (figure 3.6) which, have 0 to 1 cycle latency. Taking memory access time into account, the 16 position x matrix was replaced with 16 different variables, form x0,x1...x14,x15. This results in removing the L1 cache cycle latency and accelerating chacha_perm_generic greatly.

Chapter 4

GPU Implementation

4.1 Tools Used

The main tool used for the acceleration of Adiantum in GPU platform was CUDA toolkit and specifically v11.6.2.[22]. The CUDA toolkit provides an environment for the development and acceleration of applications on nvidia GPU-accelerated embedded systems. Developed by Nvidia, the Cuda toolkit includes, GPU-accelerated libraries, debugging and optimization tools, nvcc compiler for C/C++, and a runtime library for building applications on major architectures including x86, Arm and POWER.

4.1.1 CUDA-GDB

The cuda toolkit provides the CUDA-GDB [23], the tool for debugging CUDA applications on Linux and QNX. The tool allows debugging in real time on hardware applications, thus facilitating development on GPUs without the potential mistakes of simulation environments. CUDA-GDB is an extension of GNU project's GDB and thus contains all its features, adding new ones for debugging GPU code. Its most important aspect is allowing simultaneous debugging in both CPU and GPU within the same runtime of an application. It also provides single stepping and user set breakpoint on CUDA applications as well as the ability to inspect memory and thread specific data.

4.2 GPU Platform

The acceleration of Adiantum was implemented and tested in the JETSON XAVIER NX developer kit

4.2.1 JETSON XAVIER NX specifications

NVIDIA Jetson Xavier NX [24] delivers up to 21 TOPS of accelerating computing, achieving the performance of a SoC in a really small and compact module. It includes 384 NVIDIA CUDA® Cores, 48 Tensor Cores, 6 Carmel ARM CPUs, and two NVIDIA Deep Learning Accelerators (NVDLA) engines. Combined with over 59.7GB/s of memory bandwidth, video encoded, and decode, a large set of IOs from high-speed CSI and PCIe to low-speed I2Cs and GPIOs, and finally, low-power modes for battery-operated systems, delivering up to 14 TOPs for AI applications in as little as 10 W. For further information on the specifications of the kit 4.2.

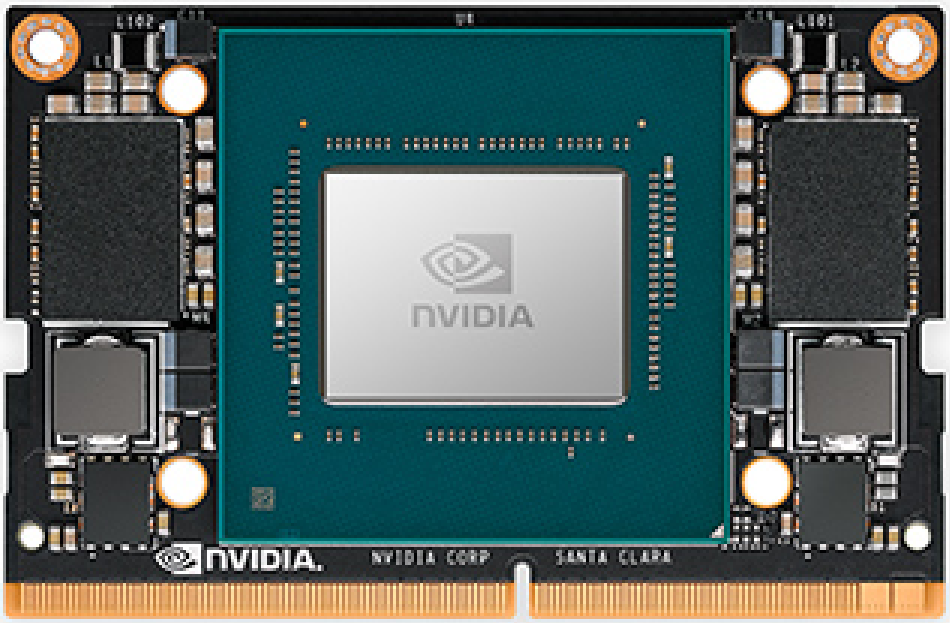

FIGURE 4.1: Jetson Xavier NX chip

	Jetson Xavier NX 16GB	Jetson Xavier NX
AI Performance	21 TOPS	
GPU	384-core NVIDIA Volta™ GPU with 48 Tensor Cores	
CPU	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6MB L2 + 4MB L3	
Memory	16 GB 128-bit LPDDR4x 59.7GB/s	8 GB 128-bit LPDDR4x 59.7GB/s
Storage	16 GB eMMC 5.1	
Power	10 W | 15 W | 20 W	
PCIe	1 x1 (PCIe Gen3) + 1 x4 (PCIe Gen4), total 144 GT/s*	
CSI Camera	Up to 6 cameras (24 via virtual channels) 14 lanes (3x4 or 6x2) MIPI CSI-2 D-PHY 1.2 (up to 30 Gbps)	
Video Encode	2x 4K60 | 4x 4K30 | 10x 1080p60 | 22x 1080p30 (H.265) 2x 4K60 | 4x 4K30 | 10x 1080p60 | 20x 1080p30 (H.264)	
Video Decode	2x 8K30 | 6x 4K60 | 12x 4K30 | 22x 1080p60 | 44x 1080p30 (H.265) 2x 4K60 | 6x 4K30 | 10x 1080p60 | 22x 1080p30 (H.264)	
Display	2 multi-mode DP 1.4/eDP 1.4/HDMI 2.0	
DL Accelerator	2x NVDLA Engines	
Vision Accelerator	7-Way VLIW Vision Processor	
Networking	10/100/1000 BASE-T Ethernet	
Mechanical	69.6 mm x 45 mm 260-pin SO-DIMM connector	

* Please refer to the Software Features section of the latest NVIDIA Jetson Linux Developer Guide for a list of supported features

FIGURE 4.2: Jetson Xavier NX Specifications

4.3 GPU distinctive features

GPUs have become an invaluable technological component for accelerating applications because of their high capacity for performing simultaneous tasks. NVIDIA, being a leading innovator for this technology has advanced their tools during the last few years, with CUDA 11 having advanced memory management and improved libraries developers. In this section some of these features will be mentioned because of their importance accelerating Adiantum.

4.3.1 The GPU Perspective

GPUs are specialized circuits designed for highly parallel processing of data, mainly intended for display purposes. Their particular ability for parallelism has made them invaluable in modern computing and thus, they are heavily used, for accelerating algorithms and AI models, even though their power consumption is really high [25]. Essentially GPUs use hundreds of different simple cores per chip working in parallel, in contrast with CPUs that have a few highly advanced and optimized computing cores. Modern CPUs are designed to handle efficiently data dependencies and branches whereas GPUs perform vastly better when a program can be highly parallelized.

It is important to mention the difference between CPUs' SIMD execution model and GPUs' SIMT. SIMD allows for a single instruction to process multiple data sets in the same clock cycle using different execution units. SIMT is similar nature but instead of using execution units, each GPU core uses several threads to perform the same instruction on different data sets, but because of GPU's contain a small per core memory, SIMT parallelism can be performed on completely different data sets simultaneously [26].

4.3.2 SIMT Approach

As mentioned in 4.3.1 GPUs have a huge capacity for parallelization. Each thread acts as if it has its own ALU, register file and memory. As a result, an embarrassingly parallel procedure can be done completely simultaneously by as many threads as the hardware provides. Specifically, the NVIDIA Volta GPU has the capacity of 1024 blocks, each containing 1024 threads [27], meaning it can perform over a million simultaneous instances of an instruction. It is important to note that GPUs are modeled to maximize data throughput, whereas CPUs are designed to perform a wider range of tasks sequentially

as fast as possible. That means that each CUDA thread is much slower than a CPU thread and the speedup is achieved by multiple threads working together. The code run by the GPU is written inside a kernel and runs asynchronously in relation to the CPU.

4.3.3 GPU memory

GPU and the CPU do not have access to the same memory by default. This problem can be solved in 2 ways:

- Use unified memory, accessible by both the CPU and the GPU.
- Transfer data form the CPU accessible memory to the GPU accessible memory.

Using unified memory, both the CPU and the GPU share a single coherent virtual memory image with a common address space [28]. At the same time they do not share the same physical memory space. This means that even if there are no explicit memory copy routines the performance in not increased. Jetson Xavier kits using CUDA 9 and above solve this problem by doing cache coherence through the the CPU cache, resulting in latency decrease and overall improved performance.

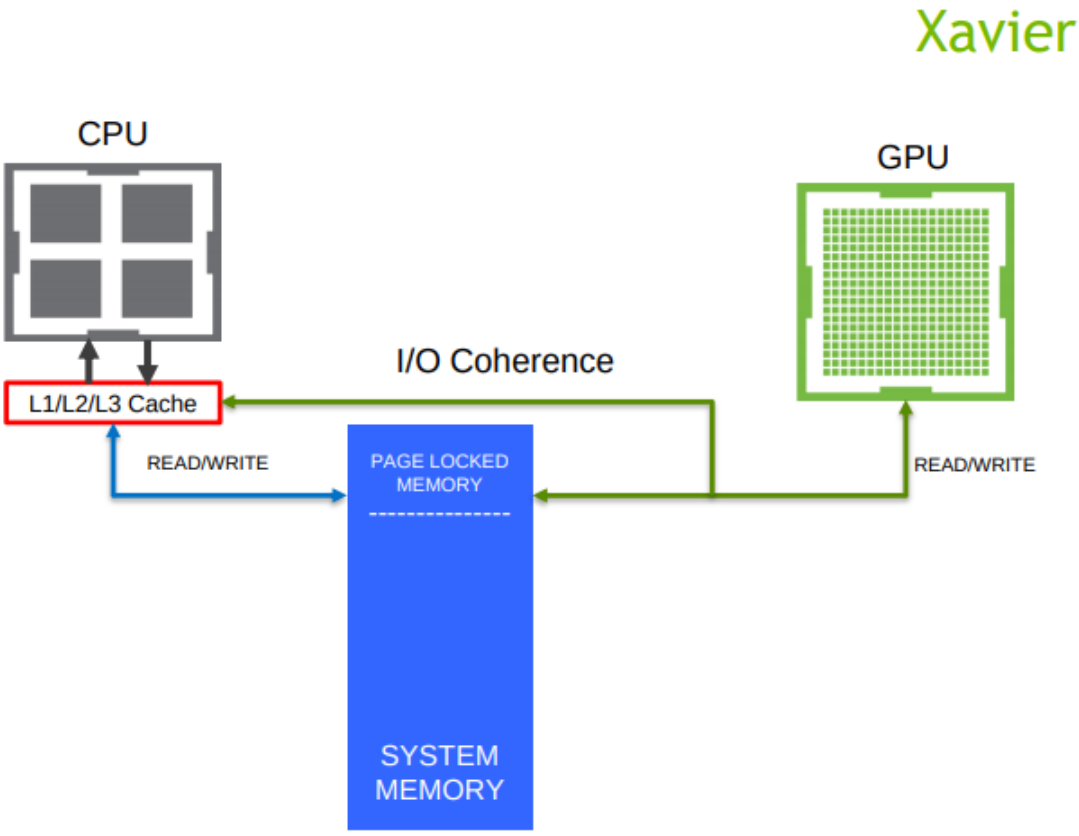

FIGURE 4.3: Cache coherence of Jetson Xavier source [29]

To use unified memory though, all variables used by the GPU need to be allocated in the unified memory first, which proved to be much slower than doing the data transfer routines once. As a result the simple memory copy routines were chosen for the sake of faster encryption.

4.4 CUDA implementation of Chacha

The approach taken by this thesis is to take the most time consuming part of the algorithm, namely XChacha stream cipher, find its most beneficial parallelization and utilize the capabilities of the Jetson GPU to do it simultaneously.

4.4.1 Chacha paralellization

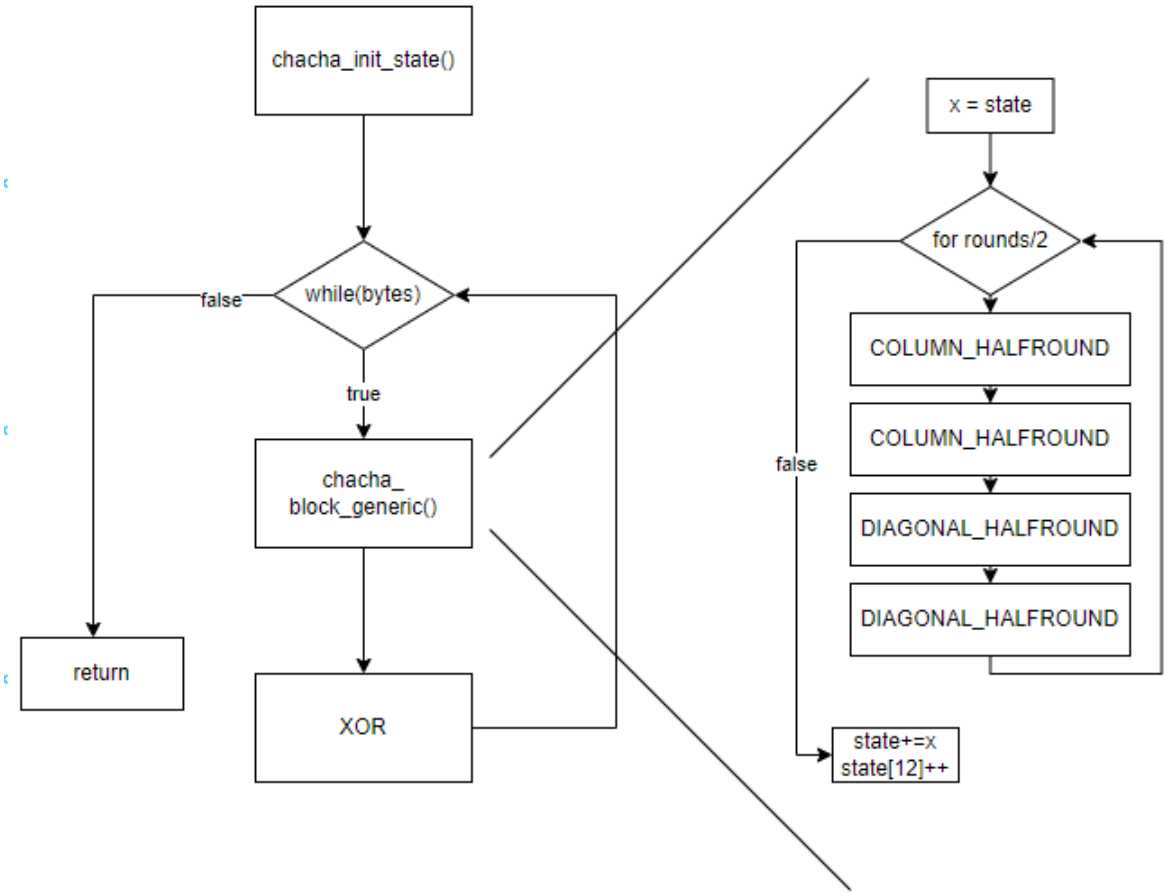

FIGURE 4.4: Flowchart of the Chacha_generic algorithm

In figure 4.4 we can see the work flow of the Chacha_generic algorithm as described in 3.4.1. Because the slowest part of the algorithm is the chacha_perm_generic loop, at first, we tried internal parallelization of the double round. The attempt proved futile because of the recursive data transfers and the fact that only four threads could work in parallel because of dependencies every four instructions within each halfround.

As a result, the parallelization was performed externally on the entire chacha_generic algorithm. This meant that each thread will encrypt one chacha block by running the chacha_block_generic algorithm and the XORing the stream with the plaintext. The procedure is embarrassingly parallel, meaning it can be executed in parallel without any race conditions.

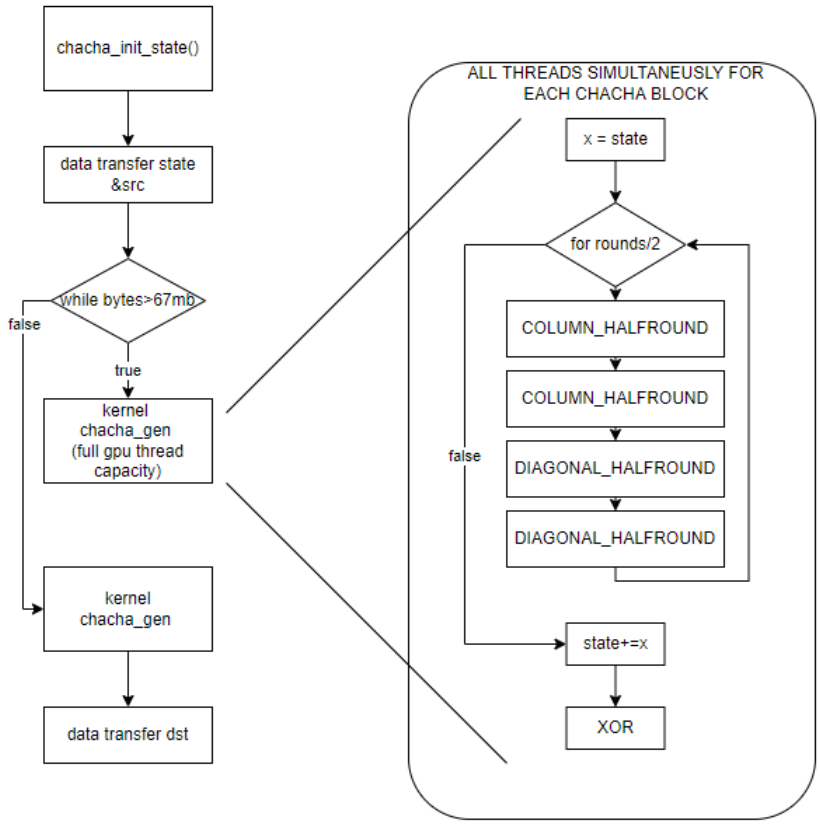

FIGURE 4.5: Flowchart of the GPU implementation of Chacha

the flowchart of the GPU optimized version of Chacha shown in figure 4.5 will be analyzed in the following subsections along with the code.

4.4.2 CUDA kernel

```
__global__ void chacha_gen(u32 stateC[16], int nrounds, u8 *dst,
                           const u8 *src, unsigned int bytes, int j) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int dim = bytes/CHACHA_BLOCK_SIZE + 1;

    if (tid<dim){
        __le32 streamC[16];
        int i;

        u32 x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x0;
        x0=stateC[0];
        x1=stateC[1];
        ...
        x12=stateC[12]+tid+j*(MAX_MESSAGE/CHACHA_BLOCK_SIZE);
        x13=stateC[13];
        x14=stateC[14];
        x15=stateC[15];
        #pragma unroll
        do {
            COLUMN_HALFROUND(16, 12, x1,x2,x3,x4,x5,x6,x7,x8,x9
                             ,x10,x11,x12,x13,x14,x15,x0);
            COLUMN_HALFROUND(8, 7,x1,x2,x3,x4,x5,x6,x7,x8,x9
                             ,x10,x11,x12,x13,x14,x15,x0);
            DIAGONAL_HALFROUND(16, 12, x1,x2,x3,x4,x5,x6,x7,x8,x9
                             ,x10,x11,x12,x13,x14,x15,x0);
            DIAGONAL_HALFROUND(8, 7, x1,x2,x3,x4,x5,x6,x7,x8,x9
                             ,x10,x11,x12,x13,x14,x15,x0);
        } while ((nrounds -= 2) != 0);

        streamC[0] = cpu_to_le32(x0 + stateC[0]);
        streamC[1] = cpu_to_le32(x1 + stateC[1]);
        ...
        streamC[12] = cpu_to_le32(x12 + stateC[12]+tid+j*(MAX_MESSAGE/CHACHA_BLOCK_SIZE));
        streamC[13] = cpu_to_le32(x13 + stateC[13]);
        streamC[14] = cpu_to_le32(x14 + stateC[14]);
        streamC[15] = cpu_to_le32(x15 + stateC[15]);

        const u8 *stream = (const u8 *)&streamC;

        size_t len = CHACHA_BLOCK_SIZE;
        #pragma unroll
        for(i=0;i<len;i++) {
            dst[tid*CHACHA_BLOCK_SIZE+i] = src[tid*CHACHA_BLOCK_SIZE+i] ^ stream[i];
            }

    }
}
```

FIGURE 4.6: CUDA kernel, chacha_generic using hardware implementation

Code in figure 4.6 shows the part of the Chacha algorithm that runs in the GPU. It is the encyrcled part of figure 4.5. The main differences between the optimized version of the original algorithm mentioned in 3.4.3 are the following.

The counter, state[12], see figure 3.2 has to increase accordingly with the block of the plaintext being encrypted, thus, the thread ID, or tid, is used

to increment the counter. Another difference is that because the XOR function of the adiantum is somewhat complicated and involves an if statement slowing the CUDA kernel down. In a kernel, every branch works as a breakpoint, making all threads wait until all previous calculations have finished, thus decreasing performance. To solve this problem, the XOR function was simplified as much as possible as shown inn the final loop. This created an issue, because streamC variable is __le32, 32 bit little endian, and both the plaintext and the ciphertext are u8, 8 bit unsigned integers. The issue was solved by creating a new variable, called stream, which is used to typecast streamC into a smaller sized variable without the loss of data. Finally all loops were unrolled via #pragma inside the kernel to avoid the previously mentioned branching issue and increase performance.

4.4.3 Kernel wrapper

```
extern "C"
void kernel_wrapper(u32 state[16], u8 *dst, const u8 *src,
                    unsigned int bytes, int nrounds)
{
    u32* stateC;
    u8* dstC;
    u8* srcC;
    int j=0;
    int NUM_THREADS;
    int NUM_BLOCKS;

    cudaMalloc((void**)&stateC, 16 * sizeof(u32));
    cudaMalloc((void**)&dstC, bytes * sizeof(u8)+CHACHA_BLOCK_SIZE);
    cudaMalloc((void**)&srcC, bytes * sizeof(const u8)+CHACHA_BLOCK_SIZE);

    cudaMemcpy(stateC, state, 16 * sizeof(u32), cudaMemcpyHostToDevice);
    cudaMemcpy(srcC, src, bytes * sizeof(const u8), cudaMemcpyHostToDevice);

    while(bytes>MAX_MESSAGE){
        NUM_THREADS = 1024;
        NUM_BLOCKS = NUM_THREADS;

        chacha_gen<<<NUM_BLOCKS, NUM_THREADS>>>
                    (stateC, nrounds, dstC, srcC, MAX_MESSAGE, j);
        bytes-=MAX_MESSAGE;
        dstC += MAX_MESSAGE;
        srcC += MAX_MESSAGE;
        j++;
        cudaDeviceSynchronize();
    }
    NUM_THREADS = sqrt(bytes/CHACHA_BLOCK_SIZE) + 1;
    NUM_BLOCKS = NUM_THREADS;

    chacha_gen<<<NUM_BLOCKS, NUM_THREADS>>>(stateC, nrounds, dstC, srcC, bytes, j);

    bytes+=j*MAX_MESSAGE;
    dstC -= j*MAX_MESSAGE;
    srcC -= j*MAX_MESSAGE;
    cudaMemcpy(dst, dstC, bytes * sizeof(u8), cudaMemcpyDeviceToHost);
    cudaFree(stateC);
    cudaFree(dstC);
    cudaFree(srcC);

}
```

FIGURE 4.7: Kernel wrapper, the function calling the CUDA kernel

Code shown in figure 4.7 corresponds to the non-encircled part of figure 4.5. As was described in 4.3.3, standard CUDA routines were used for data transfers between the CPU and the GPU. Worth mentioning is the cudaDeviceSynchronize() function which forces the GPU to finish all calculations before proceeding. This line of code is necessary, even though it decreases performance, because CUDA kernels are asynchronous and race conditions are inevitable when calling consecutive kernels. Also only one instance of cudaMalloc()

and cudaMemcpy() is used for both the ciphertext and the plaintext, regardless of size because it is faster than having consecutive calls.

Perhaps even more worth noting is the upper limit of the threads of the GPU. Jetson Xavier NX has a maximum block size of 1024 blocks, each containing 1024 threads. Each thread encrypts one Chacha block of 64 bytes. That means that if all threads work simultaneously, they will encrypt a 67MB message. The MAX_MESSAGE constant in 4.7 refers to that 67MB. For plaintexts bigger than 67MB, the kernel need run again after the previous one has finished. Thus, j variable was introduced as a counter to how many kernels have been called so as to increment the state[12] counter within the kernel. During the final kernel call, the number of blocks and threads used are exactly the one needed.

4.4.4 Linking issues

Kernel code for GPUs as well as CUDA libraries are written in C++ and need to be compiled with the nvcc compiler. At the same time Adiantum is written in C. Linking C and C++ is not a major issue for programming but the developers of Adiantum use a ninja [30] for building the algorithm. Ninja is a build system constructed to optimize speed. The problem is that ninja does not support the nvcc compiler. To solve this problem we needed to first compile the .cu files via nvcc, move those files to the build directory, and then link them manually though the ninja build file, while adding the right libraries for CUDA to run properly.

Chapter 5

Results

This chapter shows our experimental results when running Adiantum on the Jetson GPU, both from software and hardware perspectives. Also important is the demonstration power and energy consumption of each iteration of the algorithm.

5.1 Specification of Compared Platforms

This thesis was carried out on the Jetson Xavier NX, whose specifications have been presented in 4.2.1. It is important to note that even though all the results that came from the Jetson CPU, NVIDIA Carmel ARM v6.2, the Adiantum algorithm has two separate versions.

- The generic version that was optimized on the GPU.
- the SIMD version, called NEON, which is significantly faster.

All the results presented in this chapter will be compared with both versions.

5.1.1 Initialization issue

The CUDA toolkit has a really slow initialization process. The first CUDA routine called would always take about 0.2 seconds which is an unacceptable amount of time for the speedup of an application as fast as Adiantum 5.1. To solve this problem, at the start of the algorithm one instance of CudaFree(0) was put so as to initialize the CUDA libraries and not have the 0.2 second delay tamper with the results.

5.2 Throughput and Latency Speedup

In computing speedup is a term used to compare the timings of two or more systems performing the same action. It was first defined by Amdahl [31]. Although Amdahl's law was first introduced as a theoretical benchmark for the speedup of multiple processor computing, it is a fundamental formula for deducing the maximum speedup for any resource enhancement in any application.

Throughput is maximum amount of tasks that can be processed by a single system for specific amount of time. Latency represents the amount of time needed for the completion of a specific process. Both these concepts are fundamental in measuring improvements of computing systems. Speedup can be calculated in accordance to Amdahl's law:

$$Speedup = \frac{Original_Latency}{Improved_Latency} = \frac{1}{(1-p) + \frac{p}{s}} \tag{5.1}$$

- **p**: percentage of the task that was enhanced.
- **s**: the speedup given to that portion by the enhancement.

5.3 Adiantum Performance

Before analyzing the final results it is crucial to present all version of the algorithm that will be compared. Chacha8, Chacha12 and Chacha20 are exactly the same having only the number of rounds performed to differentiate them. As their name suggests 8,12,20 are their round numbers receptively. It can be deduced that the more rounds performed equals more security at the cost of execution time.

The **generic** C **version** of Adiantum that we improved, is an optimized version of the python version used in [4] and as a result comparing our GPU acceleration and their FPGA acceleration is not that useful. The **software optimization**, refers to the simple optimization described in 3.4.3. The **GPU optimization** is the main focus of this thesis and thoroughly described in the previous chapter.

Finally the **NEON version** inserts assembly intrinsics [32], taking full advantage of the SIMD benefits, achieving faster execution of the Chacha and nh-poly1305 hash functions, that represent more than 95% of the total Adiantum

processing time.

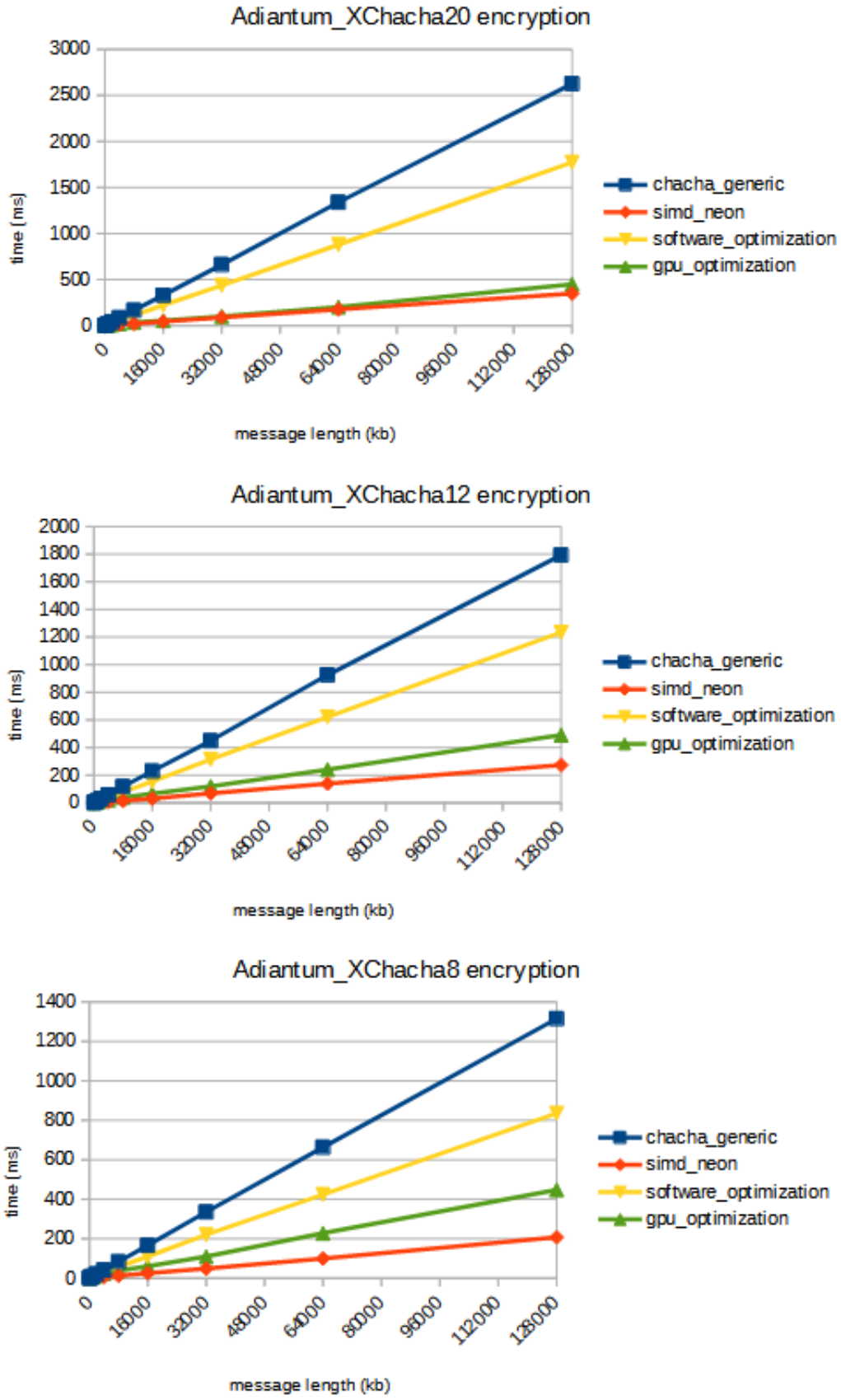

FIGURE 5.1: Performance comparison of the 4 different variations of the 3 versions of Adiantum encryption algorithm

From figure 5.1 it is clear that the execution time of the algorithm increases linearly with the plaintext size. It can also be observed that the maximum speedup is achieved in Adiantum_XChacha20 which is the slowest version. This was expected because all 20 rounds are performed simultaneously in the GPU, whereas in the CPU they are performed linearly resulting in a much greater slowdown. A more interesting note is that all three GPU implementations take about the same time, proving that in the parallel execution, the number of rounds is unimportant. That happens because the cost of 12 more rounds done once by one GPU core without any memory accesses is insignificant.

A very important remark that emerges from figure 5.1 is that, even if the GPU implementation of Adiantum_XChacha20 comes really close, the SIMT approach never reaches the speed of the SIMD approach. That happens mainly due to the fact that the NEON version has optimized both the nh-poly1305 hash functions, thus being able to reach even greater speeds than Amdahl's maximum theoretical speed described in 4.3.3. Another reason for it is the time it takes to copy large plaintexts in the GPU memory.

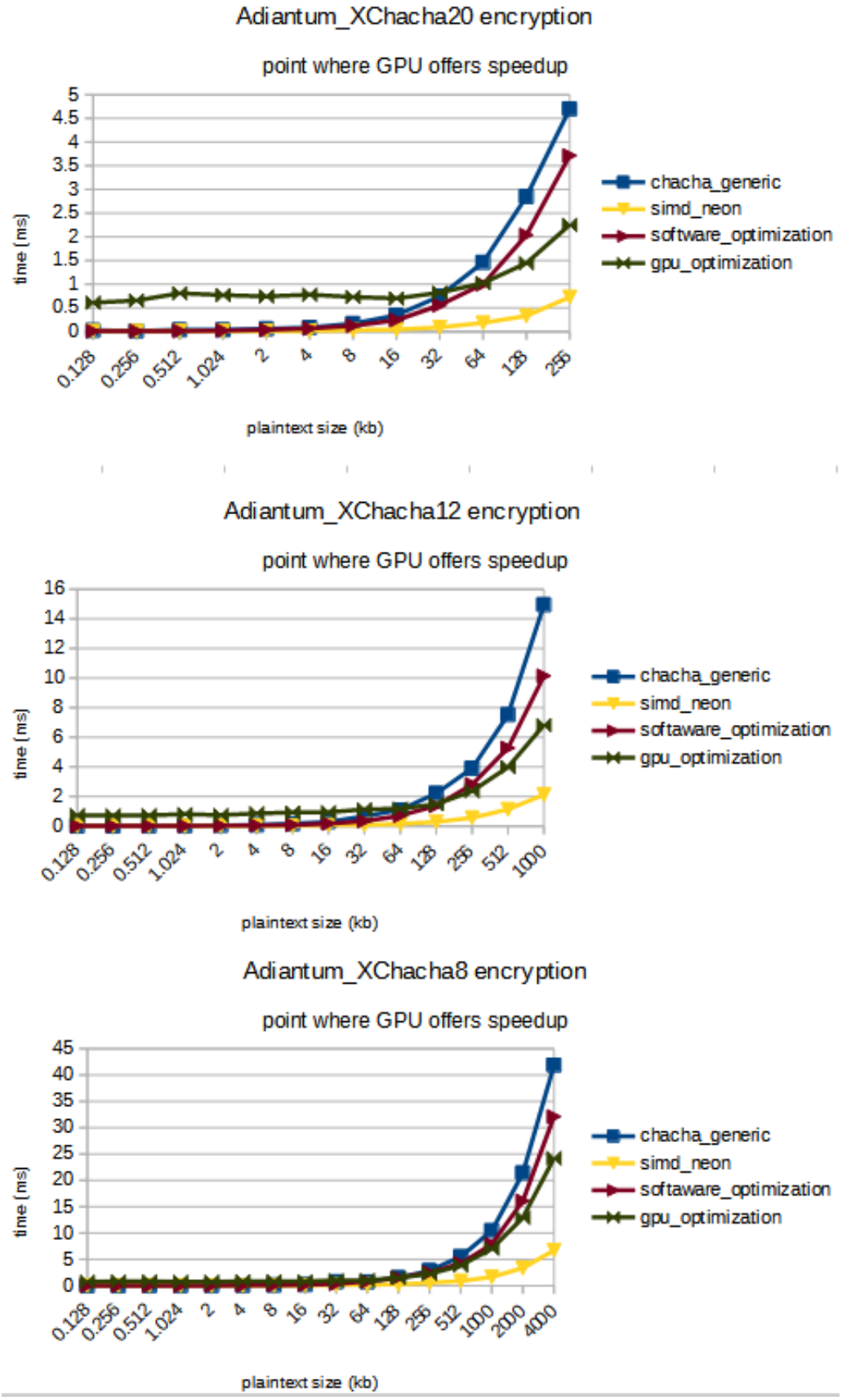

FIGURE 5.2: Point where acceleration is observed from the GPU in all versions of Adiantum_XChacha

Figure 5.2 also make it clear that speedup is observed after a certain message size, mostly because of the data transfers from and to the GPU and secondarily because single CUDA cores are weaker than the CPU core. These data transfers take between 0.6 to 0.9 ms for small data sizes.

For Adiantum_XChacha20, speedup between the GPU and the generic version is first observed for plaintext greater of equal to 64kb and for plaintext greater of equal to 128kb the GPU becomes faster than our software optimization. For Adiantum_XChacha12 and Adiantum_XChacha8 speedup between the GPU and both the generic and the optimized version is first observed for plaintext greater of equal to 128kb.

The fact that speedup is smaller or is observed in larger plaintext sizes is more than expected, since it has been stated that the GPU implementation has the same speed regardless of the number of Chacha rounds, whereas the generic/NEON/software optimized version get much faster with the reduction of Chacha rounds.

Figure 5.3 shows that NEON version of the algorithm has a much higher throughput than all other versions, which scales really fast and peaks between 64-128kb of plaintext. After that it decreases at a very slow rate. Another important observation is that the GPU implementation's throughput peaks at 32Mb where half the power of the GPU is used. That happens because very large block sizes, 1024 threads per block, may limit performance, because of resource limits (e.g. registers per thread usage, or shared memory usage) which prevent 2 threadblocks from being resident on a SM. Our software optimization always has a bigger throughput than that of the original version. The GPU implementation bests both the generic and the optimized versions at 256kb message length for Adiantum_XChacha20 and Adiantum_XChacha12, and at 4Mb for Adiantum_XChacha8. At its peak, our version has:

- 7 times the generic version's throughput for Adiantum_XChacha20.
- 4.5 times the generic version's throughput for Adiantum_XChacha12.
- 3.5 times the generic version's throughput for Adiantum_XChacha8.

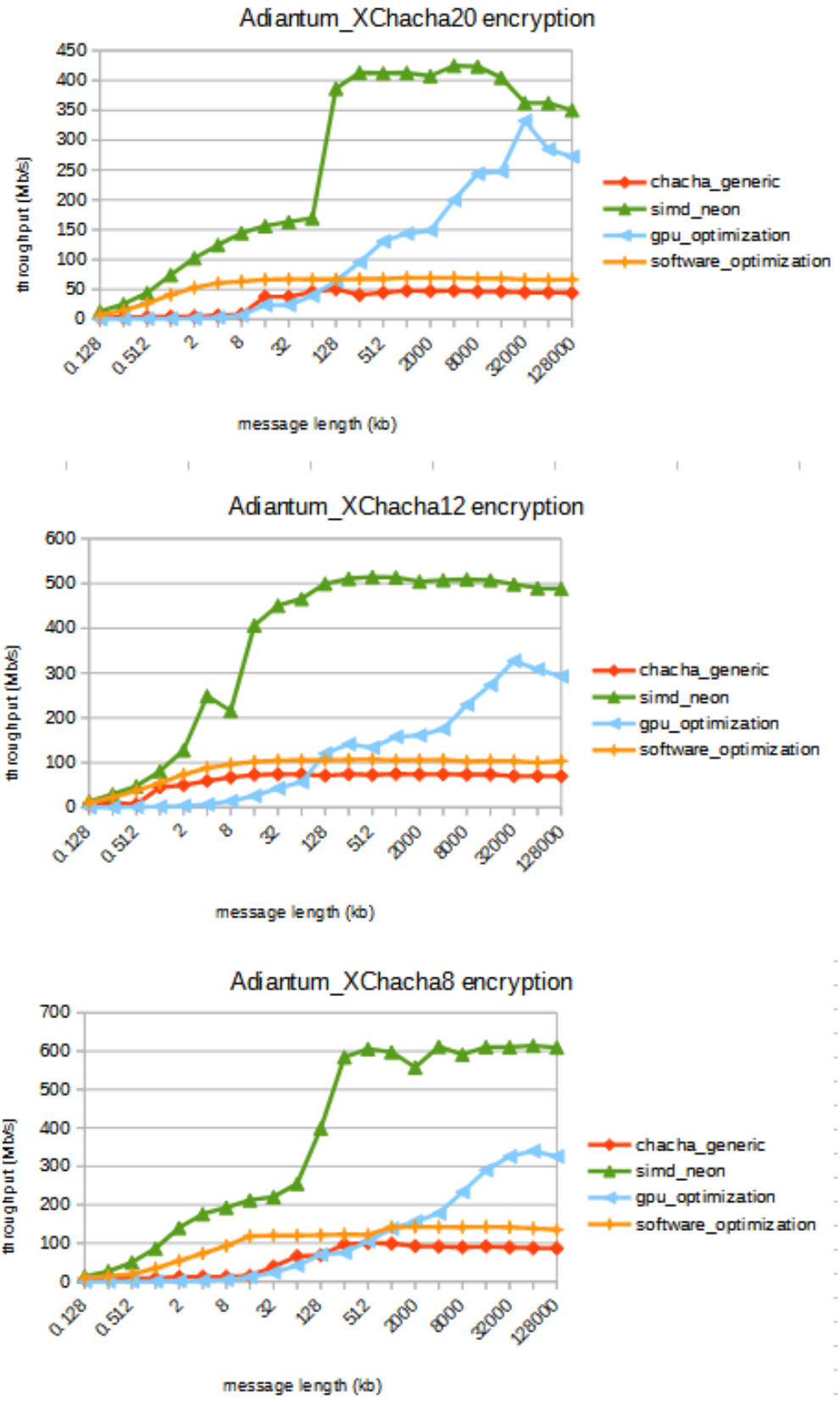

FIGURE 5.3: Throughput comparison of the 4 different variations of the 3 versions of Adiantum encryption algorithm

TABLE 5.1: XChacha20 encryption energy consumption for all 4 variations

Plaintext (kb)	Chacha generic encrypt(mJ)	Chacha NEON encrypt(mJ)	Software optimization encrypt(mJ)	GPU encrypt(mJ)
4	0.421	0.046	0.182	4.891
1000	80.032	8.670	47.056	33.267
64000	4692.863	614.582	3091.725	1086.757

TABLE 5.2: XChacha12 encryption energy consumption for all 4 variations

Plaintext (kb)	Chacha generic encrypt(mJ)	Chacha NEON encrypt(mJ)	Software optimization encrypt(mJ)	GPU encrypt(mJ)
4	0.378	0.054	0.132	4.891
1000	52.300	7.105	33.742	33.267
64000	3762.294	370.501	2208.117	1055.029

As far as energy consumption is concerned, jtop package [33] was used. It is a simple, yet very useful package, monitoring NVIDIA Jetson Developer kits. With jtop we were able to monitor the power consumed by the Jetson while running the algorithm. Energy consumption was calculated using the power consumed each second while running many iterations of the algorithm, equation 5.2 and using basic mathematical analysis integral calculus [34]. The method used has room for accuracy but unfortunately jtop does not offer more precise metrics.

$$Energy = Power * time \tag{5.2}$$

TABLE 5.3: XChacha8 encryption energy consumption for all 4 variations

Plaintext (kb)	Chacha generic encrypt(mJ)	Chacha NEON encrypt(mJ)	Software optimization encrypt(mJ)	GPU encrypt(mJ)
4	0.139	0.049	0.140	4.061
1000	36.616	5.457	26.758	34.536
64000	2621.581	370.999	1512.912	1107.569

Tables 5.1/5.2/5.3 lead us to the following results. The NEON version is by far less consuming than all other versions for all message sizes. For small plaintexts, GPU implementation is more than 10 times more energy consuming than all other iterations. For 1Mb sized messages, the generic version is about 2.5 times more energy consuming than the GPU version of Adiantum_XChacha20, about 1.5 times more energy consuming than the GPU version for Adiantum_XChacha12 and about as consuming as the GPU version for Adiantum_XChacha8. The software optimization is generally more energy efficient than the generic version and less energy efficient than the GPU implementation for large files. For large plainexts in Adiantum_XChacha20, the GPU implementation is 4.5 times less energy consuming than the generic version, 3 times less than the optimized one and 1.5 times more than the NEON version. For large plainexts in Adiantum_XChacha12, the GPU implementation is 3.5 times less energy consuming than the generic version, 2 times less than the optimized one and 2.5 times more than the NEON version. For large plainexts in Adiantum_XChacha8 the GPU implementation is 2.5 times less energy consuming than the generic version 1.5 times less than the optimized one and 2.5 times more than the NEON version.

Chapter 6

Conclusions and Future Work

This final chapter of this thesis concludes and summarizes the research done. In addition it presents possible future work on the acceleration of Adiantum cryptography algorithm in order to institute more research on the subject.

6.1 Conclusions

This thesis revolves around acceleration an the, relatively new, Adianntum cryptography algorithm using a state of the art GPU, Jetson Xavier NX. The profiling of the algorithm proved that the most time consuming part of the algorithm is the XChacha stream cipher, even when XChacha8, which is faster that its counterparts, is used. Taking advantage of Chacha algorithm's vast capacity for parallelism and GPU's multi thread capabilities we achieved the following results.

1. The GPU implementation of Adiantum_XChacha20 for large files was
    - 6 times faster and 4.5 times less energy consuming than the original algorithm for large files.
    - 4 times faster and 3 times less energy consuming than our optimized version of the original algorithm.
    - 1.15 times slower and 1.5 times more energy consuming than the SIMD NEON version of the algorithm.
2. The GPU implementation of Adiantum_XChacha20 for large files was
    - 4 times faster and 3.5 times less energy consuming than the original algorithm for large files.
    - 2.5 times faster and 2 times less energy consuming than our optimized version of the original algorithm.

- 2 times slower and 2.5 times more energy consuming than the SIMD NEON version of the algorithm.

3. The GPU implementation of Adiantum_XChacha8 for large files was
   - 3 times faster and 2.5 times less energy consuming than the original algorithm for large files.
   - 2 times faster and 1.5 times less energy consuming than our optimized version of the original algorithm.
   - 2.3 times slower and 2.5 times more energy consuming than the SIMD NEON version of the algorithm.

It is really important to note that in the GPU approach, the same speed is achieved regardless of the number of rounds of Chacha, offering the ability to upgrade security without compromising speed. That happens because of the simultaneous encryption of all Chacha blocks, making the Chacha extra rounds to be executed in parallel taking an insignificant amount of time. Unfortunately we weren't able to match the NEON version of the algorithm, neither in speed or energy consumption. We hope research on the areas presented in the next section will improve Adiantum's performance using GPU even further.

6.2 Future Work

Being relatively new, Adiantum algorithm has only K. Ampatzidis's thesis [4] as a relevant research. Adiantum's portability makes it ideal for encryption on low end devices and by combining so many cryptography models, it is very sturdy in terms of security. As a result further exploration on acceleration the algorithm will probably be considered useful in the near future.

It is clear that Adiantum is already a really fast cryptography algorithm, so it is interesting to test and analyze how much it can be optimized for very large plaintexts. Using the capabilities provided by the CUDA toolkit, the next step in acceleration Adiantum is by using streams [35]. All CUDA operations except for the kernels are synchronous, meaning the next operation is executed right after the previous has finished. Using streams it is possible to overlap, in a pipelined way the cudaMemcpy routine, which in very large messages is time consuming, with the kernel. The method is tricky though

because of the difficulty in regulating asynchronously so many threads without race conditions. By using streams we would expect an average of 20ms (20%) speedup to our GPU implementation for files of 64Mb.

Another interesting optimization would be the use of hardware to accelerate the hash function of the optimized versions of Adiantum since their time percentage in Adinatum's execution is not irrelevant anymore. Ultimately though, the importance of Adiantum is directly correlated to the time it stays secure without being compromised. That is a risk taken by all cryptography algorithms and it is the most important metric defining their relevance. Only the future will show which course Adiantum will take.

References

[1] Erik Gregersen. "Moore's law". In: *Encyclopaedia Britannica* (). URL: https://www.britannica.com/technology/Moores-law.

[2] B.J. Copeland. "Alan Turing". In: *Encyclopaedia Britannica* (). URL: https://www.britannica.com/biography/Alan-Turing.

[4] Kostantinos Ampatzidis. "Hardware Acceleration of Adiantum Cryptography Algorithm on PYNQ". Apr. 2021. URL: https://dias.library.tuc.gr/view/88981?locale=el.

[5] Paul Crowley and Eric Biggers. "Adiantum: length-preserving encryption for entry-level processors". In: *IACR Transactions on Symmetric Cryptology* (Dec. 2018), pp. 39–61. DOI: https://doi.org/10.13154/tosc.v2018.i4.39-61. URL: https://tosc.iacr.org/index.php/ToSC/article/view/7360.

[6] Kwangki Ryoo Guard Kanda. "High-Throughput Low-Area Hardware Design of Authenticated Encryption with Associated Data Cryptosystem that Uses ChaCha20 and Poly1305". In: *International Journal of Recent Technology and Engineering (IJRTE)* 8 (July 2019). URL: https://www.ijrte.org/wp-content/uploads/papers/v8i2S6/B10170782S619.pdf.

[7] Maksim Bobrov. "Cryptographic algorithm acceleration using CUDA enabled GPUs in typical system configurations". Jan. 2010. URL: https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=4203&context=theses.

[8] Weiling Cai Ziheng Wang Heng Chen. "A hybrid CPU/GPU Scheme for Optimizing ChaCha20 Stream Cipher". In: *IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking* (2021). URL: http://www.cloud-conf.net/ispa2021/proc/pdfs/ISPA-BDCloud-SocialCom-SustainCom2021-3mkuIWCJVSdKJpBYM7KEKW/264600b171/264600b171.pdf.

[9] Jake Luck Debra L. Cook John Ioannidis. "CryptoGraphics Secret Key Cryptography Using Graphics Cards". In: *Department of Computer Science Columbia University, New York, USA* (2004). URL: https://angelosk.github.io/Papers/2004/gc_ctrsa.pdf.

[10] Springer London. *Stream Ciphers*. ISBN: 978-1-4471-5079-4. DOI: https://doi.org/10.1007/978-1-4471-5079-4.

[11] Lars R. Knudsen. "Block Ciphers". In: *Encyclopedia of Cryptography and Security* (2011), pp. 152–157. DOI: 10.1007/978-1-4419-5906-5_549. URL: https://doi.org/10.1007/978-1-4419-5906-5_549.

[12] Ted Krovetz. "UMAC: Message Authentication Code using Universal Hashing". In: *RFC 4418* (Mar. 2006). URL: https://www.rfc-editor.org/rfc/rfc4418.txt.

[13] Daniel J. Bernstein. "The Poly1305-AES message-authentication code". In: *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, revised selected papers* (). URL: https://cr.yp.to/mac/poly1305-20050329.pdf.

[14] Mihir Bellare and Phillip Rogaway. "On the Construction of Variable Input-Length Ciphers". In: *Fast Software Encryption* (1999), pp. 231–244. URL: https://cseweb.ucsd.edu/~mihir/papers/lpe.pdf.

[15] Ronald L. Rivest Moses Liskov and David Wagner. *Tweakable Block Ciphers*. Advances in Cryptology — CRYPTO. 2002, pp. 31–46. ISBN: 978-3-540-45708-4.

[16] Phillip Rogaway et al. "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption". In: *Proceedings of the 8th ACM conference on Computer and Communications Security* (Nov. 2001), pp. 196–205. DOI: 10.1145/501983.502011. URL: https://dl.acm.org/doi/10.1145/501983.502011.

[17] Daniel J. Bernstein. "ChaCha, a variant of Salsa20". In: *: State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland* (Jan. 2008). URL: https://cr.yp.to/chacha/chacha-20080128.pdf.

[18] Daniel J. Bernstein. "Extending the Salsa20 nonce". In: *Workshop record of Symmetric Key Encryption Workshop 2011* (2011). URL: https://cr.yp.to/chacha/chacha-20080128.pdf.

[19] Daniel J. Bernstein. "The Salsa20 family of stream ciphers". In: *New Stream Cipher Designs: The eSTREAM Finalists* (2008). URL: https://cr.yp.to/snuffle/salsafamily-20071225.pdf.

[20] Mohit Arora. "Handling Endianness". In: *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits. New York* (2012),

pp. 155–168. DOI: 10.1007/978-1-4614-0397-5_7. URL: https://doi.org/10.1007/978-1-4614-0397-5_7.

[25] David Defour Sylvain Collange and Arnaud Tisserand. *Power Consumption of GPUs from a Software Perspective*. Computational Science – ICCS. Dec. 2009, pp. 914–923. ISBN: 978-3-642-01970-8.

[27] Mark Harris Luke Durant Olivier Giroux and Nick Stam. *Inside Volta: The World's Most Advanced Data Center GPU*. May 2017. URL: https://developer.nvidia.com/blog/inside-volta/.

[31] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. AFIPS '67* (Apr. 1967), pp. 483–485. DOI: 10.1145/1465482.1465560. URL: https://dl.acm.org/doi/10.1145/1465482.1465560.

[34] Sever Angel Popescu. "Mathematica Analysis II intergral calculus". In: *Technical University of Civil Engineering Bucharest* (Feb. 2011), pp. 1–10. URL: https://www.academia.edu/29342614/MATHEMATICAL_ANALYSIS_II_INTEGRAL_CALCULUS.

[35] Steve Rennich. *CUDA C/C++ Streams and Concurrency*. URL: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

External Links

[3] *Symmetric And Asymmetric Key Cryptography: A Detailed Guide In 2022*. URL: https://www.jigsawacademy.com/blogs/cyber-security/symmetric-and-asymmetric-key-cryptography.

[21] *Computer Memory Hierarchy*. URL: https://www.bit-tech.net/reviews/tech/memory/the_secrets_of_pc_memory_part_1/3/.

[22] *CUDA Toolkit*. URL: https://docs.nvidia.com/cuda/.

[23] *CUDA GDB*. URL: https://docs.nvidia.com/cuda/cuda-gdb/index.html.

[24] *Jetson Xavier NX Series Modules*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[26] *CUDA Tutorial*. URL: https://jhui.github.io/2017/03/06/CUDA/.

[28] *Unified Memory Programming*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[29] *I/O coherence*. URL: https://developer.ridgerun.com/wiki/index.php/File:Iocoherence.png.

[30] *ninja build*. URL: https://ninja-build.org/.

[32] *compiler intrinsics*. URL: https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?redirectedfrom=MSDN&view=msvc-170.

[33] *jtop package*. URL: https://rnext.it/jetson_stats/jtop.html.